Naïve Computational Type Theory *

Robert L. Constable

February 11, 2003

Preface

The basic concepts of type theory are fundamental to computer science, logic and mathematics. Indeed, the language of type theory connects these regions of science. It plays a role in computing and information science akin to that of set theory in pure mathematics.

There are many excellent accounts of the basic ideas of type theory, especially at the interface of computer science and logic — specifically, in the literature of programming languages, semantics, formal methods and automated reasoning. Most of these are very technical, dense with formulas, inference rules, and computation rules. Here we follow the example of the mathematician Paul Halmos, who in 1960 wrote a 104-page book called *Naïve Set Theory* intended to make the subject accessible to practicing mathematicians. His book served many generations well.

This article follows the spirit of Halmos' book and introduces type theory without recourse to precise axioms and inference rules, and with a minimum of formalism. I start by paraphrasing the preface to Halmos' book. The sections of this article follow his chapters closely.

Every computer scientist agrees that every computer scientist must know some type theory; the disagreement begins in trying to decide how much is some. This article contains my partial answer to that question. The purpose of the article is to tell the beginning student of advanced computer science the basic type theoretic facts of life, and to do so with a minimum of philosophical discourse and logical formalism. The point throughout is that of a prospective computer scientist eager to study programming languages, or database systems, or computational complexity theory, or distributed systems or information discovery.

In type theory, "naïve" and "formal" are contrasting words. The present treatment might best be described as informal type theory from a naïve point of view. The concepts are very general and very abstract; therefore they may

*This work was supported by the DoD Multidisciplinary University Research Initiative (MURI) program administered by the Office of Naval Research, under Grant N00014-01-1-0765.

take some getting used to. It is a mathematical truism, however, that the more generally a theorem applies, the less deep it is. The student's task in learning type theory is to steep himself or herself in unfamiliar but essentially shallow generalities until they become so familiar that they can be used with almost no conscious effort.

Type theory has been well exposited in articles by N. G. de Bruijn and the Automath group; the writings of Per Martin-Löf, the originator of many of the basic ideas; the writings of Jean-Yves Girard, another originator; the writings of the Coq group, the Cornell group, and the Gothenberg group; and the writings of others who have collectively expanded and applied type theory.

What is new in this account is treatment of classes and computational complexity theory along lines that seem very natural. This approach to complexity theory raises many new questions as can be seen by comparison to the lectures of Niel Jones at the summer school.

Table of Contents

1 Types and equality

Section 1 of *Naïve Set Theory* notes that the book will not define sets. Instead an intuitive idea (perhaps erroneous) will be delineated by saying what can be correctly done with sets. Halmos notes already on page one that for the purposes of mathematics one can assume that the only members of sets are other sets. *It is believed that all mathematical concepts can be coded as sets.* This simplifying approach makes set theory superficially very different from type theory.

Likewise, we delineate an intuition (possibly erroneous) about types: that intuition might be acquired from programming languages or databases or from set theory. We explain what can be correctly done with types, and how they are used.

Beginning students of computer science might believe that the only members of types need to be *bits* since all implementations ultimately reduce any data to

bits. But this level of abstraction is too low to allow a good mathematical theory. The right abstraction is at the "user level" where distinctions are made between numbers, characters, booleans and other basic kinds of data. Type theory starts in the middle, axiomatizing the user-level building blocks for computation and information. From these it is possible to define realizations as bits — going "downward" to the machine — and to define abstractions to classes, going "upward" to systems.

Elements

Types are collections of elements, possibly empty. Elements are the data. When a type is defined, the structure of the elements is specified. A type definition says how to construct elements and how to take them apart. The basic way to do this is to provide a construction pattern. On top of these, a more abstract characterization can then be given in terms of operations called *constructors* and *destructors.* But in order to define these computationally, we need at least one concrete symbolic representation of the basic data. The representation must be concrete like bits but general enough to naturally describe the objects we manipulate mentally when we calculate.

The structure of data in type theory is given by *abstract syntax.* This follows in the tradition of Lisp with its S-expressions as the basic data. To define a type, we specify the form of its data elements. To say that we have a data element is precisely to say that we have it in a specific predetermined format. This is what is critical about all implementations of data types: we must know the exact format. We call this exact format the *canonical form* of the data. To have an element is to have access to its canonical form.

Let's illustrate these ideas for the type of *bits.* The data elements we normally have in mind are 0 and 1, but let us analyze what this choice means. We could also think of bits as boolean values; then we might use *true* and *false.* We might prefer the constants of some programming language such as $0B$, $1B$.

The notion we have in mind is that there are two *distinct symbols* that represent the bits, say 0, 1, and a bit is formed from them. We can say that $bit\{0\}$, $bit\{1\}$ are the data formats, or $0B$, $1B$ are.

We intend that there are precisely two bits, given in distinct data formats. To say this precisely we need a criterion for when two canonical data elements are equal. For instance, we need to agree that we are using $bit\{0\}$, not $0B$, or agree that $bit\{0\}$ and $0B$ are equal. Defining a type means settling these matters.

Let us agree that $bit\{0\}$ and $bit\{1\}$ are the data elements formed from the distinct symbols 0 and 1. The only equalities are $bit\{0\}{=}bit\{0\}$ and $bit\{1\}{=}bit\{1\}$. These are required since equality must be an equivalence relation — hence reflexive. How do we say that $bit\{0\}{\neq}bit\{1\}$?

The answer comes from understanding what it means to take the data elements apart or *use them.* Key to our notion of the type is that 0 and 1 are *distinct* characters. This is part of the type definition process, and moreover we mean that we can *effectively* distinguish the two symbols. For example, the

operation to check for equality of bits could be to access the parts inside {} and compare them. This depends on our presumed ability to distinguish these characters. We can make the computation more abstract by creating a computational form inside the type theory rather than relying on our computational intuition about symbols so directly.

The way we accomplish the abstraction is to postulate an effective operation on the data formats. The operation is to decide whether the data is $bit\{0\}$ or $bit\{1\}$. We represent this operation by a computation rule on a syntactic form created explicitly to make decisions. Let us call the form a *conditional expression* written as

$$\textit{if } b \textit{ then } s \textit{ else } t \textit{ fi}.$$

The only meaning given to the form at this stage is in terms of the computation rules.

$$\begin{array}{ll} \textit{if } bit\{0\} \textit{ then } s \textit{ else } t \textit{ fi} & \text{reduces to } s \\ \textit{if } bit\{1\} \textit{ then } s \textit{ else } t \textit{ fi} & \text{reduces to } t \end{array}$$

With this form we can distinguish $bit\{0\}$ and $bit\{1\}$ as long as we can distinguish anything. That is, suppose we know that s≠t. Then we can conclude that $bit\{0\}\neq bit\{1\}$ as long as if-then-else-fi *respects equality.* That is, $e_1 = e_2$ should guarantee that

$$\textit{if } e_1 \textit{ then } s \textit{ else } t \textit{ fi} = \textit{if } e_2 \textit{ then } s \textit{ else } t \textit{ fi}.$$

Given this, if $bit\{0\}=bit\{1\}$, then s=t. Let's take 0 for s and 1 for t. Then since 0≠1 as characters, $bit\{0\}\neq bit\{1\}$.

Why don't we just postulate that $bit\{0\}\neq bit\{1\}$? One reason is that this can be derived from the more fundamental computational fact about if-then-else-fi. This computational fact must be expressed one way or another. We'll see later that a second reason arises when we analyze what it means to know a proposition; knowing 0≠1 is a special case.

Equality

In a sense the equality relation defines the type. We can see this clearly in the case of examples such as the integers with respect to different equalities. Recall that "the integers modulo k," $\mathbb{Z}_k$, are defined in terms of this equivalence relation, $mod(k)$, defined

$$x = y \ mod(k) \text{ iff } (x - y) = k \cdot m \text{ for some } m.$$

$\mathbb{Z}_2$ equates all even numbers, and in set theory we think of $\mathbb{Z}_2$ as the two equivalence classes (or residue classes)

$$\{0, \pm 2, \pm 4, \ldots\} \text{ and } \{\pm 1, \pm 3, \pm 5, \ldots\}.$$

It is easy to prove that $x = y \ mod(k)$ is an equivalence relation. What makes $\mathbb{Z}_k$ so interesting is that these equivalence relations are also congruences

on the algebraic structure of $\mathbb{Z}$ with respect to addition $(+)$, subtraction $(-)$, and multiplication $(*)$.

In type theory we do not use equivalence classes to define $\mathbb{Z}_k$. Instead we define $\mathbb{Z}_k$ to be $\mathbb{Z}$ with a new equality relation. We say

$$\mathbb{Z}_k == \mathbb{Z}//mod(k).$$

The official syntax is

$$\mathbb{Z}_k == quotient(\mathbb{Z}; x, y.x = y\ mod(k)).$$

This syntax treats $x, y.x + y\ mod(k)$ as a *binding construct.* The binding variables x and y have as scope the expression following the dot. In general, if A is a type and E is an equivalence relation on A, then $A//E$ is a new type such that $x = y\ in\ A//E$ iff xEy.

Recall that an equivalence relation written xEy instead of the usual relation notation $E(x, y)$ satisfies:

1.	xEx	reflexivity
2.	xEy implies yEx	commutativity
3.	xEy and yEz implies xEz	transitivity

The official syntax is quotient$(A;\ x, y.\ xEy)$.

These types, $A//E$, are called *quotient types,* and they reveal quite clearly the fact that a type is characterized by its equality. We see that $\mathbb{Z}$ and $\mathbb{Z}_2$ *have the same elements but different equalities.*

We extend the notion of membership from the canonical data to any expression in this way. If expression a' reduces to expression a, and a is a canonical element of A, then a' is an element of A by definition. For example, if $a \in A$, then *if* $bit\{0\}$ *then* a *else* b *fi* belongs to A as well.

If A is a type, and if a, b are elements of A, then $a = b\ in\ A$ denotes the equality on A. We require that equality respect computation. If $a \in A$ and a' reduces to a, then $a' = a\ in\ A$.

In set theory the membership proposition, "x is a member of y," is written $x \in y$. This proposition is specified axiomatically. It is not presented as a relation defined on the collection of all sets. Because $x \in y$ is a proposition, it makes sense to talk about *not* $x \in y$, symbolically $\neg(x \in y)$ or $x \notin y$.

In type theory, *membership is not a proposition* and not a relation on a predefined collection of all objects that divides objects into types. We write $a\,in\,A$ to mean that a is an object of type A; this is a basic judgment. It tells us what the form of object a is. If you like, A is a specification for how to build a data object of type A. To judge $a\ in\ A$ is to judge that indeed a is constructed in this way.

It does not make sense to regard $a\ in\ B$ as a proposition; for instance, it is not clear what the negation of $a\ in\ B$ would mean. To write $a\ in\ B$ we must first know that a is an object. So first we would need to know $a\ in\ A$ for some type A. That will establish what a is. Then if B is a type, we can ask whether elements of A are indeed also elements of B.

We could define a relative membership relation which is a proposition, say $x \in B$ *wrt* A, read "x belongs to B with respect to A." This means that given x is of type A, this relation is true exactly when x is also in the type B. We will have little occasion to use this predicate except as a comparison to set theory. The reasons for its scarce use are discussed when we talk about universes and open-endedness in Section 6. But we see next a proposition that addresses some useful examples of relative membership.

2 Subtypes and set types

We say that A is a *subtype* of B (symbolically, $A \sqsubseteq B$) if and only if $a = a'$ *in* A implies that $a = a'$ *in* B. Clearly $A \sqsubseteq A$, and if $A \sqsubseteq B$ and $B \sqsubseteq C$, then $A \sqsubseteq C$. For the empty type, *void*, *void* $\sqsubseteq A$ for any A.

The subtype relation induces an *equivalence relation* on types as follows. Define $A \equiv B$ if and only if $A \sqsubseteq B$ and $B \sqsubseteq A$. Clearly $A \equiv B$ is an equivalence relation, i.e. $A \equiv A$, if $A \equiv B$ then $B \equiv A$ and if $A \equiv B$ and $B \equiv C$, then $A \equiv C$. This equivalence relation is called *extensional equality*. It means that A and B have the same elements; moreover, the equality relations on A and on B are the same.

In set theory, two sets S_1, S_2 are equal iff they have the same elements. This is called *extensional equality*. Halmos writes this on page 2 as an axiom. In type theory it is a definition not an axiom, and furthermore, it is not the only equality on types. There is a more primitive equality that is *structural* (or *intensional*); we will encounter it soon.

The subtype relation allows us to talk about relative membership. Given a *in* A and given that B is a type, we can ask whether $A \sqsubseteq B$. If $A \sqsubseteq B$, then we know that a *in* A implies a *in* B, so talking about the "B-like structure of a" makes sense.

We now introduce a type that is familiar from set theory. Halmos takes it up in his section 2 under the title "axiom of specification." The axiom is also called separation. He says that:

> To every set A and to every condition $S(x)$, there corresponds a set B whose elements are exactly those elements x of A for which $S(x)$ holds.

This set is written $\{x : A \mid S(x)\}$. Halmos goes on to argue that given the set $B = \{x : A \mid x \notin x\}$, we have the curious relation

$$(*)\ x \in B \ \text{ iff } \ x \in A \ \text{ and } \ x \notin x.$$

If we assume that either $B \in B$ or $B \notin B$, then we can prove that $B \notin A$. Since A is arbitrary, this shows that either there is no set of all sets or else the law of excluded middle, P or $\neg P$, does not hold on sets. The assumption that

there is a universal set and that the law of excluded middle is true leads to the contradiction known as *Russell's paradox*.

In type theory the set type is defined just as in set theory. To every type A and relation S on A there is a type whose elements are exactly those elements of A that satisfy S. The type is denoted $\{x : A \mid S(x)\}$. But unlike in set theory, we cannot form $B = \{x : A \mid x \in x\}$ because $x \in x$ is not a relation of type theory. It is a judgment. The closest proposition would be relative membership: $x \in y \ wrt \ A$ (but y must be a type).

Also unlike set theory, we will not assume the law of excluded middle, for reasons to be discussed later. Nevertheless, we cannot have a type of all types. The reasons are deeper, and are discussed in Section 6.

Notice that we do know this:

$$\{x : A \mid S(x)\} \sqsubseteq A$$

Also we can define an empty type, *void*. We use $\emptyset$ as its display.

$$\emptyset = \{x : A \mid x \neq x \ in \ A\}.$$

We know that for any type B

$$\emptyset \sqsubseteq B.$$

In type theory, we distinguish $\{x : A \mid x = x \text{ in } A\}$ from A itself. While $A \equiv \{x : A \mid x = x \ in \ A\}$, we say that $\{x : A \mid S(x)\} = \{x : A' \mid S'(x)\}$ iff $A = A'$ and $S(x) = S'(x)$ for all x.

Here is another interesting fact about subtyping.

Theorem 1 *For all types A and equivalence relations E on A, $A \sqsubseteq A//E$.*

This is true because the elements of A and $A//E$ are the same, and since E is an equivalence relation over A, we know that E must respect the equality on A, that is

$$x = x' \ in \ A \text{ and } y = y' \ in \ A \text{ implies that } xEy \text{ iff } x'Ey'.$$

Thus, since xEx, then $x = x' \ in \ A$ implies xEx'.

3 Pairs

Halmos devotes one chapter to *unordered pairs* and another one to *ordered pairs*. In set theory ordered pairs are built from unordered ones using a clever "trick;" in type theory the ordered pair is primitive. Just as in programming, ordered pairing is a basic building block. Ordered pairs are the quintessential data elements. But unordered pairs are not usually treated as distinct kinds of elements.

Given a type A, an unordered pair of elements can be defined as:

$$\{x : A \mid x = a \vee x = b\}.$$

This is a type. We might also write it as $\{a, b\}_A$. It is unordered because $\{a, b\}_A \equiv \{b, a\}_A$. We'll see that this notion does not play an interesting role in type theory. It does not behave well as a data element, as we see later.

Given types A, B, their *Cartesian product,* $A \times B$, is the type of ordered pairs, $pair(a; b)$. We abbreviate this as $< a, b >$. Given $a\ in\ A$, $b\ in\ B$, then $< a, b >\ in\ A \times B$. The constructor $pair(a; b)$ structures the data. The obvious destructors are operations that pick out the first and second elements:

$$1of(< a, b >) = a \qquad 2of(< a, b >) = b.$$

These can be defined in terms of a single operator, $spread()$, which splits a pair into its parts. The syntax of $spread()$ involves the idea of *binding* variables. They are used as a pattern to describe the components. Here is the full syntax and the rules for computing with it.

> If p is a pair, then $spread(p; u, v.g)$ describes an operator g for decomposing it as follows:
>
> $$spread(< a, b >; u, v.g) \text{ reduces in one step to } g[a/u, b/v]$$
>
> where $g[a/u, b/v]$ is the result of substituting a for the variable u, and b for the variable v.
>
> Define
>
> $$1of(p) == spread(p; u, v.u)$$
> $$2of(p) == spread(p; u, v.v).$$
>
> Notice that
>
> $$spread(< a, b >; u, v.u) \text{ reduces to } a$$
> $$spread(< a, b >; u, v.v) \text{ reduces to } b.$$

Is there a way to treat $\{a, b\}$ as a data element analogous to $< a, b >$? Can we create a type $Pair(A; B)$ such that

$$\{a, b\} \text{ in } Pair(A; B)$$
$$\{a, b\} = \{b, a\} \text{ in } Pair(A; B) \text{ and } Pair(A; B) = Pair(B; A)?$$

If we had a union type, $A \cup B$, we might define the pairs as $\{a, b\} = \{x : A \cup B | (x = a\ in\ A) \text{ or } (x = b\ in\ B)\}$, and let $Pair(A; B)$ be the collection of all such pairs. We exploe this collection later, in Sections 4 and 6.

How would we use $\{a, b\}$? The best we can do is pick elements from the pair, say $pick(p; u, v.g)$ and allow either reduction:

$$pick(\{a, b\}; u, v.g) \text{ reduces either to } g[a/u, b/v] \text{ or to } g[b/u, a/v].$$

4 Union and intersection

Everyone is familiar with taking unions and intersections of sets and writing them in the standard cup and cap notations respectively, as in the union $X \cup Y$ and intersection $X \cap Y$. These operations are used in type theory as well, with the same notations. But the meanings go beyond the set idea, because the definitions must take account of equalities.

Intersection is the easier idea. If A and B are types, then equality holds on $A \cap B$ when it holds in A and in B; that is,

$$a = b \, in \, A \cap B \quad \text{iff} \quad a = b \, in \, A \quad \text{and} \quad a = b \, in \, B.$$

In particular,

$$a = a \, in \, A \cap B \quad \text{iff} \quad a = a \, in \, A \quad \text{and} \quad a = a \, in \, B.$$

For example, $\mathbb{Z}_2 \cap \mathbb{Z}_3$ has elements such as 0, since $0 = 0$ in $\mathbb{Z}_2$ and $0 = 0$ in $\mathbb{Z}_3$. And $0 = 6$ holds in $\mathbb{Z}_2 \cap \mathbb{Z}_3$ since $0 = 6$ in $\mathbb{Z}_2$ and $0 = 6$ in $\mathbb{Z}_3$. In fact, $\mathbb{Z}_2 \cap \mathbb{Z}_3 = \mathbb{Z}_6$.

Intersections can be extended to families of types. Suppose $B(x)$ is a type for every x in A. Then $\cap x{:}A.\ B(x)$ is the type such that

$$b = b' \, in \, \cap x{:}A.\ B(x) \text{ iff } b = b' \, in \, B(x) \text{ for all } x \, in \, A.$$

For example, if $\mathbb{N}_k = \{0, \ldots, k-1\}$ and $\mathbb{N}^+ = \{1, 2, \ldots\}$ then $\cap x : \mathbb{N}^+.\mathbb{N}_x$ has only 0 in it.

It is interesting to see what belongs to $\cap x : A.\ B(x)$ if A is empty. We can show that there is precisely one element, and any closed expression of type theory denotes that element. We give this type a special name because of this interesting property.

Definition $Top == \cap x{:}void.\ x$, for *void*, the empty type.

Theorem 2 *If A, A' are types and $B(x)$ is a family of types over A, then*

1. $A \cap A' \sqsubseteq A \quad A \cap A' \sqsubseteq A'$
2. $\cap x{:}A.\ B(x) \sqsubseteq B(a)$ *for all a in A*
3. $A \sqsubseteq Top$
4. *If $C \sqsubseteq A$ and $C \sqsubseteq A'$, then $C \sqsubseteq A \cap A'$*

If A and B are disjoint types, say $A \cap B = void$, then their *union*, $A \cup B$, is a simple idea, namely

$$a = b \text{ in } A \cup B \text{ iff } a = b \text{ in } A \text{ or } a = b \text{ in } B.$$

In general we must consider equality on elements that A and B have in common. The natural thing to do is extend the equality so that if $a = a'$ in A and $a' = b$ in B, then $a = b$ in $A \cup B$. Thus the equality of $A \cup B$ is the *transitive closure* of the two equality relations, i.e.

$$a = b \ in \ A \cup B \text{ iff} \qquad \begin{array}{l} a = b \ in \ A \text{ or } a = b \ in \ B \quad or \\ \exists\, c : A \cup B \quad a = c \ in \ A \cup B \text{ and } c = b \ in \ A \cup B. \end{array}$$

Note in $\mathbb{Z}_2 \cup \mathbb{Z}_3$ all elements are equal, and $\mathbb{Z}_4 \cup \mathbb{Z}_6 = \mathbb{Z}_2$.

Exercise: What is the general rule for membership in $\mathbb{Z}_m \cup \mathbb{Z}_n$?

Exercise: Unions can be extended to families of types. Give the definition of $\cup x : A.\ B(x)$. See the Nuprl Web page under basic concepts, unions, for the answer.

Theorem 3 *If A and A' are types and $B(x)$ is a family of types over A, then*

1. $A \sqsubseteq A \cup A' \qquad A' \sqsubseteq A \cup A'$
2. $B(a) \sqsubseteq \cup x : A.\ B(x)$ *for all a in A*
3. *If $A \sqsubseteq C$ and $A' \sqsubseteq C$, then $A \cup A' \sqsubseteq C$.*

In set theory the disjoint union $A \oplus B$ is defined by using tags on the elements to force disjointness. We could use the tags *inl* and *inr* for the left and right disjuncts respectively. The definition is

$$A \oplus B = \{< inl, a >, < inr, b > \mid a \in A, b \in B\}.$$

The official definition is a union:

$$(\{inl\} \times A) \cup (\{inr\} \times B).$$

We could define a disjoint union in type theory in a similar way. Another approach that is more common is to take disjoint union as a new primitive type constructor, $A + B$.

If A and B are types, then so is $A + B$, called their disjoint union. The elements are $inl(a)$ for $a \ in \ A$ and $inr(b)$ for $b \ in \ B$.

The destructor is $decide(d; u.g_1; v.g_2)$, which reduces as follows:

$$decide(inl(a); u.g_1; v.g_2) \text{ reduces to } g_1[a/u] \text{ in one step}$$

$$decide(inr(b); u.g_1; v.g_2) \text{ reduces to } g_2[b/v] \text{ in one step.}$$

Sometimes we write just $decide(d; g_1; g_2)$, if g_1 and g_2 do not depend on the elements of A and B, but only on the tag. In this way we can build a type isomorphic to Bit by forming $Top + Top$.

5 Functions and relations

We now come to the heart of type theory, an abstract account of *computable functions over all types.* This part of the theory tells us what it means to have an effectively computable function on natural numbers, lists, rational numbers, real numbers, complex numbers, differentiable manifolds, tensor algebras, streams, on any two types whatsoever. It is the most comprehensive theory of effective computability in the sense of deterministic sequential computation. *Functions are the main characters of type theory in the same way that sets are the main characters of set theory and relations are the main characters of logic.* Types arise because they characterize the domains on which an effective procedure terminates.

The computational model is more abstract than machine models, say Turing machines or random access machines or networks of such machines. Of all the early models of computability, this account is closest to the idea of a high-level programming language or the lambda calculus. A distinguishing feature is that all of the computation rules are independent of the type of the data, e.g., they are *polymorphic.* Another distinguishing feature is that the untyped computation system is *universal* in the sense that it captures at least all effective sequential procedures — terminating and non-terminating.

Functions

If A and B are types, then there is a type of the effectively computable functions from A to B, and it is denoted $A \to B$. Functions are also data elements, but the operations on functions do not expose their internal structure. We say that the canonical form of this data is $\lambda(x.\, b)$. The symbol lambda indicates that the object is a function. (Having any particular canonical form is more a matter of convenience and tradition, not an essential feature.) The variable x is the formal input, or *argument.* It is a binding variable that is used in the *body* of the function, b, to designate the input value. The *body* b is the *scope* of this binding variable x. Of course the exact name of the variable is immaterial, so $\lambda(x.\, b)$ and $\lambda(y.\, b[y/x])$ are equal canonical functions.

The important feature of the function notation is that the body b is an expression of type theory which is *known to produce a value of type B after finitely many reduction steps* provided an element of type A, say a, is input to the function. The precise number of reduction steps on input a is the *time complexity* of $\lambda(x.\, b)$ on input a. We write this as

$$b[a/x] \downarrow b' \text{ in } n \text{ steps}$$

Recall that $b[a/x]$ denotes the expression b with term a substituted for all free occurrences of x. A free occurrence of x in b is one that does not occur in any subexpression c which is in the scope of a binding occurrence of x, i.e. not in $\lambda(x.\, c)$ or $spread(p; u, v.c)$, where one of u, v is x. The notion of binding occurrence will expand as we add more expressions to the theory. The above

definition applies, of course, to all future extensions of the notion of a binding occurrence.

The simplest example of a computable function of type $A \to A$ is the identity function $\lambda(x.\ x)$. Clearly if $a \in A$, then $x[a/x]$ is $a \in A$.

If a_0 is a constant of type A, then $\lambda(x.\ a_0)$ is the constant function whose value is a_0 on any input. Over the natural numbers, we will have functions such as $\lambda(x.\ x+1)$, the successor, $\lambda(x.\ 2*x)$, doubling, etc.

We say that two functions, f and g, are *equal on* $A \to B$ when they produce the same values on the same inputs.

$$(f = g\ in\ A \to B) \text{ iff } f(a) = g(a)\ in\ B \text{ for all } a\ in\ A.$$

This relation is called *extensional equality.* It is not the only sensible equality, but it is the one commonly used in mathematics.

We might find a tighter (finer) notion of equality more useful in computing, but no widely agreed-upon concept has emerged. If we try to look closely at the structure of the body b, then it is hard to find the right "focal length." Do we want the details of the syntax to come into focus or only some coarse features of it, say the combinator structure?

Functions are not meant to be "taken apart" as we do with pairs, nor do we directly use them to make distinctions, as we use the bits 0, 1. Functions are encapsulations of computing procedures, and the principal way to use them is to *apply them.* If f is a function, we usually display its application as $f(a)$ or fa. To conform to our uniform syntax, we write application primitively as $ap(f;a)$ and display this as $f(a)$ or fa when no confusion results.

The computation rule for $ap(f;a)$ is to first reduce the expression for f to canonical form, say $\lambda(x.\ b)$, and then to reduce $ap(\lambda(x.\ b);a)$. One way to continue is to reduce this to $b[a/x]$ and continue. Another way is to reduce a, say to a', and then continue by reducing $b[a'/x]$. The former reduction method is called *call by name* and the latter is *call by value.* We will use both kinds, writing $apv(f;a)$ for call by value.

Notice that when we use call by name evaluation, the constant function $\lambda(x.\ a_0)$ maps B into A for *any* type B, even the void type. So if $a_0 \in A$, then $\lambda(x.\ a_0) \in B \to A$ for any type B.

The function $\lambda(x.\ \lambda(y.x))$ has the type $A \to (B \to A)$ for any types A and B, regardless of whether they are empty or not. We can see this as follows. If we assume that z is of type A, then $ap(\lambda(x.\ \lambda(y.\ x));z)$ reduces in one step to $\lambda(y.\ z)$. By what we just said about constant functions, this belongs to $(B \to A)$ for any type B. Sometimes we call the function $\lambda(x.\ \lambda(y.x))$ the K *combinator.* This stresses its polymorphic nature and indicates a connection to an alternative theory of functions based on combinators.

Exercise: What is the general type of these functions?

1. $\lambda(x.\ \lambda(y.\ <x,y>))$
2. $\lambda(f.\ \lambda(g.\ \lambda(x.\ g(x)(f(x)))))$

The function in (1) could be called the *Currying combinator,* and Curry called the function in (2) the *S combinator;* it is a form of composition.

The function $\lambda(f.\lambda(g.\lambda(x.g(f(x)))))$ belongs to the type $(A \to B) \to ((B \to C) \to (A \to C))$ because for $x \in A$, $f(x) \in B$ and $g(f(x)) \in C$. This is a *composition combinator, Comp.* It shows clearly the polymorphic nature of our theory. We can express this well with intersection types:

$$Comp\, in\, (\cap A, B, C : Type.\ (A \to B) \to ((B \to C) \to (A \to C))).$$

We will need to discuss the notion $A : Type$ etc. in Section 6 before this is entirely precise.

There are polymorphic λ-terms that denote sensible computations but which cannot be directly typed in the theory we have presented thus far. One of the most important examples is the so-called *Y-combinator* discovered by Curry,

$$\lambda(f.\ \lambda(x.f(xx))\lambda(x.f(xx))).$$

The subtyping relation on $A \to B$ behaves like this:

Theorem 4 *For all types* $A \sqsubseteq A',\ B \sqsubseteq B'$

$$A' \to B \sqsubseteq A \to B'.$$

To see this, let $f = g \in A' \to B$. We prove that $f = g\ in\ A \to B'$. First notice that for $a \in A$, we know $a \in A'$, thus $f(a')$ and $g(a')$ are defined, and $f(a') = g(a')$ in B. But $B \sqsubseteq B'$, so $f(a') = g(a')$ in B'.

This argument depends on the polymorphic behavior of the functions; thus, if $f(a)$ terminates on any input in A', it will, as a special case, terminate for any element of a smaller type. We say that $\sqsubseteq$ is *co-variant* in the domain.

In set theory, functions are defined as *single-valued relations,* and relations are defined as sets of ordered pairs. This reduces both relations and functions to sets, and in the process the reduction obliterates any direct connection between functions and algorithms.

A function $f \in A \to B$ does generate a type of ordered pairs called its *graph*, namely

$$graph(f) = \{x : A \times B | f(1of(x)) = 2of(x)\ in\ B\}.$$

Clearly $graph(f) \sqsubseteq A \times B$, somewhat as in set theory. If we said that a relation R on $A \times B$ is any subtype of $A \times B$, then we would know that $R \sqsubseteq A \times B$. But we will see that in type theory we cannot collect all such R in a single "power type." Let us see how relations are defined in type theory.

Relations

Since functions are the central objects of type theory, we define a *relation* as a certain kind of function, a logical function in essence. This is Frege's idea of a relation. It depends on having a type of propositions in type theory. For

now we denote this type as $Prop$, but this is amended later to be $Prop_i$ for fundamental reasons.

A relation on A is a *propositional function* on A, that is, a function $A \to Prop$. Here are propositional functions we have already encountered:

$$x = y \; in \; A$$

is an atomic proposition of the theory. From it we can define two propositional functions:

$$\lambda(x.\ \lambda(y.\ x = y \; in \; A)) \;\; in \; A \to (A \to Prop)$$

$$\lambda(p.\ 1of(p) = 2of(p) \; in \; A) \;\; in \; A \times A \to Prop.$$

We also discussed the proposition

$$x \in B \; wrt \; A,$$

from which we can define the propositional function

$$\lambda(x.\ x \in B \; wrt \; A) \; in \; A \to Prop \text{ for a fixed } B.$$

This propositional function is well-defined iff

$$x = y \; in \; B \text{ implies that } (x \in B \; wrt \; A)$$

is equal to the proposition

$$(y \in B \; wrt \; A).$$

The type $Prop$ includes propositions built using the *logical operators* & (and), $\vee$ (or), $\Rightarrow$ (implies), and $\neg$ (not), as well as the *typed quantifiers* $\exists\, x : A$ (there is an x of type A) and $\forall x : A$ (for all x of type A). We mean these in the *constructive sense* (see Section 9).

6 Universes, powers and openness

In set theory we freely treat sets as objects, and we freely quantify over them. For example, the ordered pair $< x, y >$ in set theory is $\{x, \{x, y\}\}$. We freely form nested sets, as in the sequence starting with the empty set: $\emptyset, \{\emptyset\}, \{\{\emptyset\}\}, \{\{\{\emptyset\}\}\}, \ldots$ All of these are distinct sets, and the process can go on indefinitely in "infinitely many stages." For example, we can collect all of these sets together

$$\{\emptyset, \{\emptyset\}, \{\{\emptyset\}\}, \{\{\{\emptyset\}\}\}, \ldots\}$$

and then continue $\{\{\emptyset, \{\emptyset\}, \ldots\}\}$, $\{\{\{\emptyset, \{\emptyset\}, \ldots\}\}\}$. There is a precise notion called the *rank* of a set saying how often we do this.

In set theory we are licensed to build these more deeply nested sets by various axioms. The axiom to justify $\{\emptyset, \{\emptyset\}, \{\{\emptyset\}\}, \{\{\{\emptyset\}\}\}, \ldots\}$ is the *axiom of infinity.* Another critical axiom for nesting sets is called the *power set* axiom.

It says that for any set x, the set of all its subsets exists, called $\mathcal{P}(x)$, the power set of x. One of its subsets is $\{x\}$; another is $\emptyset$.

In type theory we treat types as objects, and we can form $\{\emptyset\}, \{\{\emptyset\}\}, \ldots$ But the licensing mechanism is different, and types are not much used as objects of computation, because more efficient data is available and is almost always a better choice.

The licensing mechanism for building large sets is critical because paradoxes arise if sets are allowed to be "too large," as with a set of all sets, called a *universe*. A key idea from set theory is used to justify large types. The idea is this: if we allow a fixed number of ways of building sets safely, such as unions, separation, and power sets, then we can form a safer kind of universe defined by allowing the iteration of all these operations indefinitely. This is the idea of a universe in type theory.

A universe is a collection of types closed under the type-building operations of pairing, union, intersection, function space and other operations soon to be defined. But it is not closed under forming universes (though a controlled way of doing this is possible), and we keep careful track of how high we have gone in this process. As a start, we index the universes as $\mathbb{U}_1, \mathbb{U}_2, \mathbb{U}_3, \ldots$ The elements are codes that tell us precisely how to build types. We tend to think of these indices as the types themselves.

Here is how we can use a universe to define $\{\emptyset\}$. Recall that $\emptyset$, the void type, could be defined as $\{x : \mathbb{B} \mid x \neq x \ in \ \mathbb{B}\}$. This type belongs to $\mathbb{U}_i$ for all i. Now define $\{\emptyset\}$ as $\{x : \mathbb{U}_1 \mid x = \emptyset \ in \ \mathbb{U}_1\}$. In our type theory, we cannot define this set without mentioning $\mathbb{U}_1$. Thus we are forced to keep track of the universe through which a type of this sort is admitted.

With universes we can define a *limited power type,* namely

$$\mathcal{P}_i(A) = \{x : \mathbb{U}_i | x \sqsubseteq A\}.$$

These might be seen as "pieces" of some ideal $\mathcal{P}(A)$ for any type A. But we do not know a provably safe way to define the ideal $\mathcal{P}(A)$.

The universe construction forms a *cumulative hierarchy,* $\mathbb{U}_1 \in \mathbb{U}_2, \mathbb{U}_2 \in \mathbb{U}_3, \ldots$, and $\mathbb{U}_i \sqsubseteq \mathbb{U}_{i+1}$. Note that $Top \, in \, \mathbb{U}_i$ for all i, and yet $\mathbb{U}_i \sqsubseteq Top$. There is nothing like Top in set theory.

7 Families

Unions, intersections, products and function spaces can all be naturally extended from pairs of types to whole families. We saw this already in the case of unions. Halmos devotes an entire chapter to families.

Let us consider the disjoint union of a family of types indexed by a type A, that is, for every x in A, there is a type $B(x)$ effectively associated to it. The disjoint union in both set theory and type theory is denoted by $\Sigma x : A.\ B(x)$, and the elements are pairs $< a, b >$ such that $b \, in \, B(a)$. In type theory this is a new primitive type constructor, but it uses the pairing operator associated with products. It also uses $spread(p; u, v.g)$ to decompose pairs. Unlike pairing

for products where $A \times B$ is empty if either A or B is, the disjoint union is not empty unless all of the $B(x)$ for each x in A are empty.

In set theory the disjoint union of a family of types is defined in terms of $A \times \cup x\!:\!A.\ B(x)$,

$$\Sigma x\!:\!A.\ B(x) = \{p : A \times \cup x\!:\!A.\ B(x) | 1of(p) \in A\ \ \&\ \ 2of(p) \in B(1of(p))\}$$

and where the ordinary union $\cup x\!:\!A.\ B(x)$ is defined as the union of the range of the function B from A into sets, e.g. $\cup\{B(x)|x \in A\}$.

The type $\Sigma x : A.\ B(x)$ behaves both like a union and a product. It is a product because its elements are pairs, so we expect the type to be related to $A \times B$. But it is not like a product in that it can be nonempty even if one of the $B(x)$ types is empty.

One reason for computer scientists to think of this as a product is that it is called a *variant record* in the literature of programming languages, and records are treated as products. It is a "variant" record because the initial componenents (leftmost) can influence the type of the later ones, causing them to "vary."

As a product, we call this type a *dependent product*, and we employ notation reminiscent of products, namely

$$x : A \times B(x)$$

where x is a binding variable, bound to be of type A and having *scope* $B(x)$. So

$$y : A \times B(y)$$

is the same type as (equal to) $x : A \times B(x)$.

Here is an example of this type. We'll consider which alternative view is most natural. Suppose the index set (first component) is in $\mathbb{N}^2 + \mathbb{R}^2$; thus it is either a pair of natural numbers or a pair of computable reals. Suppose the second component will be a number representing the area of a rectangle defined by the first pair. Its type will be $\mathbb{N}$ or $\mathbb{R}$, depending on the first value. To define this, let $Area(x)$ be defined as

$$Area(x) = decide(x; \mathbb{N}; \mathbb{R})$$

(or, more informally, $Area(x) = \textit{if}\ \ is\_left(x)\ \ \textit{then}\ \mathbb{N}\ \ \textit{else}\ \mathbb{R}\ \ \textit{fi}$). The type we want is $\Sigma x\!:\!(\mathbb{N}^2 + \mathbb{R}^2).\ Area(x)$, or equivalently $x : (\mathbb{N}^2 + \mathbb{R}^2) \times Area(x)$. How do we think of this? Is it a union of two types $(\mathbb{N}^2 \times \mathbb{N})$ and $(\mathbb{R}^2 \times \mathbb{R})$, or does it describe data that looks like $< inl(< nat, nat >), nat >$ or $< inr(< real, real >), real >$?

There are examples where the contrast is clearer. One of the best is the definition of dates as $< month, day >$, where

$$\begin{array}{rcll} Month & = & \{1, \ldots, 12\} & \text{and} \\ Day(2) & = & \{1, \ldots, 28\} & \\ Day(i) & = & \{1, \ldots, 30\} & \text{for } i = 3, 6, 9, 11 \\ Day(i) & = & \{1, \ldots, 31\} & \text{otherwise.} \end{array}$$

$Date = m : Month \times Day(m)$ seems like the most natural description of the data we use to represent dates, and the idea behind it is clearly a pair whose second component depends on the first.

We can also extend the function space constructor to families, forming a *dependent function space*. Given the family $B(x)$ indexed by A, we form the type

$$x : A \to B(x).$$

The elements are functions f such that for each $a \in A$, $f(a)\ in\ B(a)$.

In set theory the same type is called an *infinite product* of a family, written $\Pi x{:}A.\ B(x)$, and defined as

$$\{f : A \to \cup x{:}A.\ B(x) |\ \forall x{:}A.\ f(x) \in B(x)\}.$$

An example of such a function takes as input a month, and produces the maximum day of the month — call it *maxday*. It belongs to

$$m : Month \to Day(m).$$

The intersection type also extends naturally to families, yet this notion was only recently discovered and exploited by Kopylov, in ways that we illustrate later. Given type A and family B over A, define

$$x : A \cap B(x)$$

as the collection of elements x of A such that x is also in $B(x)$. If $a = b$ in A and $a = b$ in $B(a)$, then $a = b$ in $x : A \cap B(x)$.

8 Lists and numbers

The type constructors we have examined so far all build finite types from finite types. The list constructor is not this way. The type $List(A)$ is limitless or *infinite* if there is at least one element in A. Moreover, the type $List(A)$ is *inductive*. From $List(A)$ we can build the natural numbers, another inductive type. These types are an excellent basis for a computational understanding of the infinite and induction.

In set theory induction is also important, but it is not as explicitly primitive; it is somewhat hidden in the other axioms, such as the axiom of infinity—which explicitly provides $\mathbb{N}$—and the axiom of regularity, which provides for (transfinite) induction on sets by asserting that every ε-chain $x_1 \in x_0,\ x_2 \in x_1,\ x_3 \in x_2, \ldots$ must terminate.

If A is a type, then so is $List(A)$. The canonical data of $List(A)$ is either the empty list, *nil*, or it is a list built by the basic *list constructor*, or *cons* for short. If L is a list, and $a\ in\ A$, then $cons(a; L)$ is a list. The standard way to show such an inductive construction uses the pattern of a rule,

$$\frac{a\ in\ A \quad L\ in\ List(A)}{cons(a; L)\ in\ List(A).}$$

Here are some lists built using elements $a_1, a_2, a_3, \ldots$ The list *nil* is in any type $List(A)$ for any type A, empty or not. *So* $List(A)$ *is always nonempty.* Next, $cons(a_1; nil)$ is a list, and so are

$$cons(a_1; nil), cons(a_1; cons(a_1; nil)), cons(a_1; cons(a_2; nil)),$$

and so forth. We typically write these as

$$[a_1], [a_1, a_2], [a_1, a_2, a_3].$$

If A is empty, then $List(A)$ has only *nil* as a member.

Here is a particularly clear list type. Let $\mathbf{1}$ be the type with exactly one element, 1. Then the list elements can be enumerated in order, nil, $cons(1; nil), cons(1; cons(1; nil)), cons(1; cons(1$ We will define the natural numbers to be this type $List(\mathbf{1})$. The type $List(Top)$ is isomorphic to this since Top has just one element, but that element has a limitless number of canonical names.

The method of destructing a list must make a distinction between *nil* and $cons(a; L)$. So we might imagine an operator like *spread*, say $dcons(L; g_1; u, t.g_2)$ where $dcons(nil; g_1; u, v.g_1)$ reduces to g_1 in one step and $dcons(cons(a; L); g_1, u, t.g_2)$ reduces to $g_2[a/u, L/t]$ in one step. This allows us to disassemble one element. The power of lists comes from the *inductive pattern* of construction,

$$\frac{a\ in\ A \quad L\ in\ List(A)}{cons(a; L)\ in\ List(A).}$$

This pattern distinguishes $List(A)$ from all the other types we built, making it infinite. We need a destructor which recognizes this inductive character, letting us apply *dcons* over and over until the list is eventually *nil*.

How can we build an inductive list destructor? We need an inductive definition for it corresponding to the inductive pattern of the elements. But just decomposing a list will be useless. We want to leave some trace as we work down into this list. That trace can be the corresponding construction of another object, perhaps piece by piece from the inner elements of the list.

Let's imagine that $build(L)$ is constructing something from L as it is decomposed. Then the inductive pattern for building something in B is just this:

$$\text{build } b_0\ in\ B \qquad \frac{a\ in\ A, \text{assume } b\ in\ B \text{ is built from } L}{\text{combine } a, L, \text{ and } b \text{ to build } g(a, L, b)\ in\ B.}$$

This pattern can be expressed in a recursive computation on a list

$$build(nil) = b_0 \qquad build(cons(a; L)) = g(a, L, build(L)).$$

This pattern can be written as a simple recursive function if we use *dcons* as follows:

$$f(L) = dcons(L; b_0; u, t.g(u, t, f(t)).$$

There is a more compact form of this expression that avoids the equational form. We extend *dcons* to keep track of the value being built up. The form is $list\_ind(L; b_0; u, t, v.g(u, t, v))$, where v keeps track of the value.

We say that $list\_ind(L; b_0; u, t, v.g(u, t, v))$ *in* B, provided that b_0 *in* B and assuming that $u \in A, t \in List(A)$ and v *in* B, then $g(u, t, v)$ *in* B.

The reduction rule for *list_ind* is just this:

$list\_ind(nil; b_0; u, t, v.g(u, t, v))$	reduces to b_0
$list\_ind(cons(a; L); b_0; u, t, v.g(u, t, v))$	reduces to $g(a, L, v_0)$

where $v_0 = list\_ind(L; b_0; u, t, v.g(u, t, v))$.

There are several notations that are commonly adopted in discussing lists. First, we write $cons(a; l)$ as $a.l$. Next we notice that

$$list\_ind(l; a; u, t, v.b)$$

where v does not occur in b is actually just a case split on whether l is nil followed by a decomposition of the cons case. We write this as:

$$\text{case of } l;\ nil \to b;\ a.t \to b$$

Here are some of the basic facts about lists along with a sketch of how we prove them.

Fact: $nil \neq cons(a; L)$ in $List(A)$ for any a or L.

This is because if $nil = cons(a; L)$ in $List(A)$, then $list\_ind(x; 0; u, t, v.1)$ would reduce in such a way that $0 = 1$, which is a contradiction. The basic fact is that all expressions in the theory respect equality.

One of the most basic operations on lists is *appending* one onto another, say $[a_1, a_2, a_3]@[a_4, a_5] = [a_1, a_2, a_3, a_4, a_5]$. Here is a recursive definition:

$$x@y = \text{ case of } x; nil \to y; a.t \to a.(t@y).$$

This abbreviates

$$list\_ind(x; \lambda(y.y); a, t, v.\lambda(y.a.v(y))).$$

Fact: For all x, y, z in $List(A)$, $(x@y)@z = x@(y@z)$.

We prove this by induction on x. The base case is $(nil@x)@y = nil@(x@y)$. This follows immediately from the nil case of the definition of @.

Assuming that $(t@x)@y = t@(x@y)$, then $((u.t)@y)@z = u.t@(y@z)$ again follows by the definition of @ in the cons case. This ends the proof.

If f is a function from A to B, then here is an operation that applies it to every element of l in $List(A)$:

$$map(f; l) = \text{case of } l;\ nil \to nil;\ a.t \to f(a).map(f; t).$$

Fact: If $f : A \to B$ and $g : B \to C$ and $l : List(A)$, then $map(g; map(f; l))$ $= map(\lambda(x.g(f(x))); l)$ in $List(C)$.

Fact: If $f : A \to B$ and $x, y : List(A)$, then $map(f; x@y) = map(f; x) @map(f; y)$ in $List(B)$.

Fact: $x@y = nil$ in $List(A)$ iff $x = nil$ and $y = nil$ in $List(A)$.

Exercise: What would $List(\{nil\})$ look like? Do all the theorems still work in this case? How does this compare to $List(\emptyset)$?

9 Logic and the Peano axioms

Following the example of Halmos, we have avoided attention to logical matters, but there is something especially noteworthy about logic and type theory. *We use logical language in a way that is sensitive to computational meaning.* For instance, when we say $\exists x : A.\ B(x)$, read as "we can find an x of type A such that $B(x)$," we mean that to know that this proposition is true is to be able to exhibit an object a of type A, called the *witness*, and *evidence* that $B(a)$ is true; let this evidence be $b(a)$. It turns out that we can think of a proposition P as *the type of evidence for its truth.* We require of the evidence only that it carry the *computational content* of the sense of the proposition. So in the case of a proposition of the form $\exists x : A.\ B(x)$, the evidence must contain a witness a and the evidence for $B(a)$. We can think of this as a pair, $< a, b(a) >$.

When we give the type of evidence for a proposition, we are specifying the computational content. Here is another example. When we know P implies Q for propositions P and Q, we have an effective procedure for taking evidence for P into evidence for Q. So the computational content for P *implies* Q is the function space $P \to Q$, where we take P and Q to be the types of their evidence.

This computational understanding of logic diverges from the "classical" interpretation. This is especially noticeable for statements involving *or.* To know $(P\ or\ Q)$ is to either know P or know Q, and to know which. The rules of evidence for $(P\ or\ Q)$ behave just as the rules for elements of the disjoint union type, $P + Q$.

The computational meaning of $\forall x : A.\ B(x)$ is that we can exhibit an effective method for taking elements of A, a, to evidence for $B(a)$. If $b(a)$ is evidence for $B(a)$ given any element of a in A, then $\lambda(x.\ b(x))$ is computational evidence for $\forall x : A.\ B(x)$. So the evidence type behaves just as $x : A \to B(x)$.

For atomic propositions like $a = b$ in B, there is no interesting computational content beyond knowing that when a and b are reduced to canonical form, they will be identical, e.g. $bit\{0\} = bit\{0\}$ or $bit\{1\} = bit\{1\}$. For the types we will discuss in this article, there will be no interesting computational content in the equality proposition, even in $f = g\ in\ A \to B$. We say that the computational content of $a = b\ in\ A$ is *trivial.* We will only be concerned with whether there is evidence. So we take some atomic object, say is_true, to be the evidence for any true equality assertion.

What is remarkably elegant in this account of computational logic is that the rules for the evidence types are precisely the expected rules for the logical

operators. Consider, for example, this computational interpretation of $\exists x : A.\neg Q(x)$ *implies* $(\forall x\!:\!A.\ (P(x) \vee Q(x))$ *implies* $\exists x\!:\!A.\ P(x)$.

To prove $\exists x\!:\!A.\ P(x)$, let a be the element of A such that $\neg Q(a)$. We interpret $\neg S$ for any proposition S as meaning S implies $False$, and the evidence type for $False$ is empty. Taking a for x in $\forall x : A.(P(x) \vee Q(x))$ we know $P(a) \vee Q(a)$. So there is some evidence $b(a)$ which is either evidence for $P(a)$ or for $Q(a)$. We can analyze $b(a)$ using *decide* since $P(a) \vee Q(a)$ is like $P(a) + Q(a)$. If $b(a)$ is in $P(a)$ then we have the evidence needed. If $b(a)$ is in $Q(a)$, then since we know $Q(a) \to False$, there is a method, call it f, taking evidence for $Q(a)$ into $False$.

To finish the argument, we look at the computational meaning of the assertion ($False$ *implies* S) for any proposition S. The idea is that $False$ is empty, so there is no element that is evidence for $False$. This means that if we assume that x is evidence for $False$, we should be able to provide *evidence for any proposition whatsoever.*

To say this computationally, we introduce a form $any(x)$ with the typing rule that if x is of type $False$, then $any(x)$ is of type S for any proposition S.

Now we continue the argument. Suppose $b(a)$ is in $Q(a)$. Then $f(b(a))$ is in $False$, so $any(f(b(a)))$ is in $P(a)$. Thus in either case of $P(a)$ or $Q(a)$ we can prove the $\exists x\!:\!A.\ P(a)$. Here is the term that provides the computation we just built:

$$\begin{aligned}\lambda(e.\lambda(all.\ & decide(all(1of(e));\\ & p.p;\\ & q.any(f(q))))).\end{aligned}$$

Note that e is evidence for $\exists x : A.\neg Q(x)$, so $1of(e)$ belongs to A. The function *all* produces the evidence for either $P(a)$ or $Q(a)$, so $all(1of(e))$ is what we called $b(a)$. We use the *decide* form to determine the kind of evidence $b(a)$ is; in one case we call it p and in the other q. So $any(f(q))$ is precisely what we called $any(f(b(a)))$ in the discussion.

Now we turn to using this logic. In 1889 Peano provided axioms for the natural numbers that have become a standard reference point for our understanding of the properties of natural numbers. We have been using $\mathbb{N}$ to denote these numbers $(0, 1, 2, \dots)$.

Set theory establishes the adequacy of its treatment of numbers by showing that the set of natural numbers, ω, satisfies the five Peano axioms. These axioms are usually presented as follows, where $s(n)$ is the *successor* of n.

1. 0 is a natural number, $0 \in \mathbb{N}$.
2. If $n \in \mathbb{N}$, then $s(n) \in \mathbb{N}$.
3. $s(n) = s(m)$ implies $n = m$.
4. Zero has no predecessor, $\neg(s(n) = 0)$.
5. The induction axiom, if $P(0)$, and if $P(n)$ implies $P(s(n))$, then P holds for all natural numbers.

In set theory axioms 1 and 2 are part of the definition of the axiom of infinity; axiom 3 is a general property of the successor of a set defined as $s(x) = x \cup \{x\}$. Induction comes from the definition of ω as the least inductive subset of the postulated infinite set.

In type theory we also deal with axioms 1 and 2 as part of the definition. One way to treat $\mathbb{N}$ is to define it as a new type whose canonical members are $0, s(0), s(s(0))$, and so forth, say using these rules:

$$0\ in\ \mathbb{N} \qquad \frac{n\ in\ \mathbb{N}}{s(n)\ in\ \mathbb{N}.}$$

Another approach is to define $\mathbb{N}$ using lists. We can take $\mathbb{N}$ as $List(\mathbf{1})$, with nil as 0 and $cons(1; n)$ as the successor operation. Then the induction principle follows as a special case of list induction.

In this definition of $\mathbb{N}$, the addition operation $x + y$ corresponds exactly to $x@y$, e.g. $2 + 3$ is just $[1, 1]@[1, 1, 1]$.

Exercise: Show how to define multiplication on lists.

10 Structures, records and classes

Bourbaki's encyclopedic account of mathematics begins with set theory, and then treats the general concept of a *structure*. Structures are used to define algebraic structures such as monoids, groups, rings, fields, vector spaces, and so forth. They are also used to define topological structures and order structures, and then these are combined to provide a modular basis for real analysis and complex analysis.

Structures

The idea of a structure is also important in computer science; they are the basis for modules in programming languages and for classes and objects in object-oriented programming. Also, just as topological, order-theoretic, and algebraic structures are studied separately and then combined to explain *aspects* of analysis, so also in computing, we try to understand separate aspects of a complex system and then combine them to understand the whole.

The definition of structures in set theory is similar to their definition in type theory, as we next illustrate; later we look at key differences. In algebra we define a *monoid* as a structure $< M, op, id >$, where M is a set, op is an associative binary operation on M, and id is a constant of M that behaves as an identity, e.g.

$$x\ op\ id = x \ \text{ and } \ id\ op\ x = x.$$

A *group* is a structure $< G, op, id, inv >$ which extends a monoid by including an inverse operator, e.g.

$$x\ op\ inv(x) = id \text{ and } inv(x)\ op\ x = id.$$

In algebra, a group is considered to be a monoid with additional structure. These ideas are naturally captured in type theory in nearly the same way. We start with a structure with almost no form,

$$A : \mathbb{U}_i \times Top.$$

This provides a carrier A and a "slot" for its extension. We can extend by refining Top to have structure, for example, $B \times Top$. Notice that $B \times Top \sqsubseteq Top$, and

$$\begin{array}{l} A \times (B \times Top) \sqsubseteq A \times Top, \quad \text{since} \\ A \sqsubseteq A \text{ and } B \times Top \sqsubseteq Top. \end{array}$$

Generally,

$$A \times (B \times (C \times Top)) \sqsubseteq A \times (B \times Top).$$

We can define a monoid as a dependent product:

$$M : \mathbb{U}_i \times ((M \to (M \to M)) \times (M \times Top)).$$

An element has the form $< M, < op, < id, \bullet >>>$ where:

$$M \; in \; \mathbb{U}_i, \;\; op \; in \; M \to (M \to M), \; id \in M, \text{ and } \bullet \; in \; Top.$$

Call this structure $Monoid$.

A *group* is an extension of a monoid which includes an inverse operator, $inv : G \to G$. So we can define the type $Group$ as

$$G : \mathbb{U}_i \times (G \to (G \to G) \times (G \times ((G \to G) \times Top))).$$

It is easier to compare these dependent structures if we require that the type components are related. So we define *parameterized* structures. We specify the *carrier* C of a monoid or group, etc., and define

$$\begin{array}{lcl} Monoid(C) & = & C \to (C \to C) \times (C \times Top) \\ Group(C) & = & C \to (C \to C) \times (C \times (C \to C \times Top)). \end{array}$$

Then we know

Fact: If $M \in \mathbb{U}_i$, then $Group(M) \sqsubseteq Monoid(M)$.

Exercise: Notice that $G \sqsubseteq M$ need not imply that

$$Group(G) \sqsubseteq Monoid(M).$$

These subtyping relationships can be extended to richer structures such as rings and fields, though not completely naturally. For example, a ring consists of two related structures: a group part (the *additive* group) and a monoid part (the *multiplicative* monoid). We can combine the structures as follows, putting the group first:

$$Ring(R) = R \to (R \to R) \times ((R \times (R \to R) \times (R \to (R \to R) \times (R \times Top)))).$$

We can say that $Ring(R) \sqsubseteq Group(R)$, but it is not true that the multiplicative structure is directly a substructure of $Monoid(R)$. We need to project it off,

$$Mult\_Ring(R) = (R \to (R \to R) \times (R \times Top)).$$

Then we can say $Mult\_Ring(R) \sqsubseteq Monoid(R)$. Given a ring

$$Rng\, in\, Ring(R),$$

$$< R, < add\_op, < add\_id, < add\_inv, < mulop, < mulid, \bullet >>>>>>,$$

we can project out the multiplicative part and the additive part:

$$\begin{array}{rcl} add(Rng) & = & < add\_op, < add\_id, < add\_inv, - >>> \\ mul(Rng) & = & < mulop, < mulid, \bullet >>, \end{array}$$

and we know that $add(Rng)\, in\, Group(R)$ and $mul(Rng)\, in\, Monoid(R)$.

Records

Our account of algebraic structure so far is less convenient than the informal one on which it is based. One reason is that we must adhere to a particular ordering of the components and access them in this order. Programming notations deal with this inconvenience by associating names with the components and accessing them by name. The programming construct is called a *record type*; the elements are *records*. A common notation for a record type is this:

$$\{a : A;\ b : B;\ c : C\}.$$

Here A, B, C are types and a, b, c are names called *field selectors*. If r is a record, the notations $r.a, r.b, r.c$ select the component with that name. The order is irrelevant, so $\{b : B;\ c : C;\ a : A\}$ is the same type, and we know that $r.a \in A, r.b \in B, r.c \in C$.

In this notation, one type for a group over G is

$$\{add\_op : G \to (G \to G);\ add\_id : G;\ add\_inv : G \to G\}.$$

The field selectors in this example come from a type called *Label*, but more generally we can say that a family of these group types abstracted over the field selectors, say $GroupType(G; x, y, z)$, is

$$\{x : G \to (G \to G);\ y : G;\ z : G \to G\}.$$

We can combine $MonoidType(G; u, v) = \{u : G \to (G \to G); v : G\}$ with the $GroupType$ to form a $RingType(R; x, y, z, u, v) =$

$$\{x : G \to (G \to G);\ y : G;\ z : G \to G;\ u : G \to (G \to G); v : G\}.$$

Record types can be defined as function spaces over an index type such as *Label*. First we associate with each x in *Lable* a type, say $T : Label \to \mathbb{U}_i$. If we have in mind a type such as group, parameterized by G, then we map *add_op* to $G \to (G \to G)$, *add_id* to G, *add_inv* to $G \to G$, and all other lables to Top. Call this map $Grp(x)$. Then the group type is

$$x : Label \to Grp(x).$$

An element of this type, g, is a function such that $g(add\_op) \in G \to (G \to G)$, $g(add\_id) \in G$, $g(add\_inv) \in G \to G$, and $g(z) \in Top$ for all other z in *Label*.

The record types we use have the property that only finitely many labels are mapped to types other than Top. For example, $\{a{:}A;\ b{:}B;\ c{:}C\}$ is given by a map $T : Label \to \mathbb{U}_i$ such that $T(a) = A$, $T(b) = B$, $T(c) = C$, and $T(x) = Top$ for x not in $\{a, b, c\}$. Thus

$$\{a : A;\ b : B;\ c : C\} = x : Label \to T(x).$$

We say that such records have *finite support* of $\{a, b, c\}$ in *Label*. If I is the finite support for T, we sometimes write the record as $x : I \to T(x)$.

It is very interesting that for two records having finite support of I_1 and I_2 such that $I_1 \subseteq I_2$, and such that $T : Label \to \mathbb{U}_i$ agree on I_2, we know that

$$x : I_2 \to T(x) \sqsubseteq x : I_1 \to T(x).$$

For example, consider $\{a, b, c\} \subseteq \{a, b, c, d\}$, with records $R_1 = \{a : A; b : B; c{:}C\}$ and $R_2 = \{a{:}A;\ b{:}B;\ c{:}C;\ d{:}D\}$. Then $R_2 \sqsubseteq R_1$. This natural definition conforms to programming language practice and mathematical practice. We will see that this definition, while perfectly natural in type theory, is not sensible in set theory.

If we use standard labels for algebraic operations on monoids, groups, and rings, say

$$Alg : \{add\_op, add\_id, add\_inv, mul\_op, mul\_id\} \to \mathbb{U}_1$$

then

$$\begin{array}{lcl} Add\_Monoid(G) & = & i : \{add\_op, add\_id\} \to Alg(i) \\ Group(G) & = & i : \{add\_op, add\_id, add\_in\} \to Alg(i) \\ Mul\_Monoid(G) & = & i : \{mul\_op, mul\_id\} \to Alg(i) \\ Ring(G) & = & i : \{add\_op, add\_id, add\_inv, mul\_op, mul\_id\} \to Alg(i) \end{array}$$

and we have

$$\begin{array}{lcl} Group(G) & \sqsubseteq & Add\_Monoid(G) \\ Ring(G) & \sqsubseteq & Group(G) \sqsubseteq Add\_Monoid(G) \\ Ring(G) & \sqsubseteq & Mul\_Monoid(G). \end{array}$$

The reason that these definitions don't work in set theory is that the subtyping relation on function spaces is not valid in set theory. Recall that the relation is

$$\frac{A \sqsubseteq A' \quad B \sqsubseteq B'}{A' \to B \sqsubseteq A \to B'.}$$

This is true in type theory because functions are *polymorphic.* If $f \in A' \to B$, then given $a \in A$, the function f applies to a; so $f(a)$ is well-defined and the result is in B, hence in B'. In set theory, the function f is a set of ordered pairs, e.g. $f = \{< a, b > \in A' \times B | f(a) = b\}$. This set of ordered pairs can be larger than $\{< a, b > \in A \times B' | f(a) = b\}$, so $A' \to B \not\subseteq A \to B'$. The difference in the notion of function is fundamental, and it is not clear how to reconcile them.

Dependent Records

A full account of algebraic structures must include the axioms about the operators. For a monoid we need to say that *op* is associative, say

(1) $Assoc\,(M, op)$ is
$\forall x, y, z : M.\ (x\ op\ y) op\ z = x\ op (y\ op\ z)\ in\ M.$

and we say that *id* is a two-sided identity:

(2) $Id(M, op, id)$ is
$\forall x : M.\ (x\ op\ id = x\ in\ M)$ and $(id\ op\ x = x\ in\ M)$.

For the group inverse the axiom is

(3) $Inv(M, op, id, inv)$ is
$\forall x : M.\ (x\ op\ inv(x) = id\ in\ M)$ and $(inv(x) op\ x = id\ in\ M)$.

In set theory, these axioms are not included inside the algebraic structure because the axioms are propositions, which are "logical objects," not sets. But as we have seen, propositions can be considered as types. So we can imagine an account of a *full-monoid* over M that looks like this:

$$\{op : G \to (G \to G);\ id : G;\ ax1 : Assoc(G, op);\ ax2 : Id(G, op, id)\}.$$

If g is a full-monoid, then $g(op)$ is the operator and $g(ax1)$ is the computational evidence that *op* is associative; that is, $g(ax1)$ is a mathematical object in the type $Assoc(G, op)$.

Does type theory support these kinds of records? We call them *dependent records*, since the type $Assoc(G, op)$ depends on the object $g(op)$. Type theory does allow us to define them.

The object we define is the general dependent record of the form

$$\{x_1 : A_1; x_2 : A_1(x_1); \ldots; x_n : A_{n-1}(x_1, \ldots, x_{n-1})\}.$$

In Automath these are called *telescopes,* and they are primitive concepts. We define them from the dependent intersection following Kopylov. Other researchers have added these as new primitive types, and Jason Hickey defined them from a new primitive type called the very-dependent function space. (See Bibliographic Notes for this section.)

Recall that the dependent intersection, $x : A \cap B(x)$ is the collection of those elements a of A which are also in $B(a)$.

We define $\{x : A; y : B(x)\}$ as the type

$$f : \{x : A\} \cap \{y : B(f.x)\}.$$

The elements of $\{x : A\}$ are the functions $\{x\} \rightarrow A$; that is, functions in $i : Label \rightarrow A(i)$ where $A(x) = A$ and $A(i) = Top$ for $i \neq x$. The singleton label, $\{x\}$, is the finite support. The elements of the intersection are those functions $f\ in\ \{x\} \rightarrow A$ such that on input y from $Label$, $f(y)\ in\ B(f(x))$.

To define $\{x : A; y : B(x); z : C(x, y)\}$, we have the choice of associating the dependent intersection to the right or left; we chose to the left.

$$*g : (f : \{x : A\} \cap \{y : B(f(x))\}) \cap \{z : C(g(x), g(y))\}.$$

The function g must agree with f on label x. We can see that the outermost binding, g, satisfies the inner constraints as well. So as we intersect in more properties, we impose more constraints on the function.

The value of associating to the left is that we can think of building up the dependent record by progressively adding constraints. It is intuitively like this:

$$(\{x : A\} \cap \{y : B(x)\}) \cap \{z : C(x, y)\}.$$

This kind of notational simplicity can be seen as an abbreviation of

$$** \ s : (s : \{x : A\} \cap \{y : B(s(x))\}) \cap \{z : C(s(x), s(y))\},$$

because the scoping rules for binding operators tell us that * and ** are equal types.

In programming languages such as ML, modules and classes are used to modularize code and to make the code more abstract. For example, our treatment of natural numbers so far has been particularly concrete. We introduced them as lists of a single atomic object such as 1. Systems like HOL take the natural numbers ($\mathbb{N}$) as primitive, and Nuprl takes the integers ($\mathbb{Z}$) as primitive, defining $\mathbb{N}$ as $\{z : \mathbb{Z} | 0 \leq z\}$. All these approaches can be subsumed using a class to axiomatize an abstract structure. We define numbers, say integers, abstractly as a class over some type D which we axiomatize as an *ordered discrete integral domain*, say $Domain(D)$. The class is a dependent record with structure. We examine this in more detail here.

First we define the stucture without induction and then add both recursive definition and induction. We will assume *display forms* for the various field selectors when we write axioms. Here is a table of displays:

Field Selector Name	**Display** (inside the class and outside)
add	+
zero	0
minus	-
mult	*
one	1
div	÷
mod	*mod infix*
less_eq	≤

The binary operators are given by the type $BinaryOp(D)$. This can be the "curried style" type $D \to (D \to D)$ that we used previously, or the more "first-order" style, $D \times D \to D$, or it can even be the "Lisp-style," where we allow any list of arguments from D, say $List(D) \to D$. We can define monoids, groups, etc. using the same abstraction, which we select only at the time of implementation. Likewise, we can abstract the notion of a binary relation to $BinaryRel(D)$. We can use $D \to (D \to Prop_i)$ or $D \times D \to Prop_i$ or $List(D) \to Prop_i$.

Definition For D a type in $\mathbb{U}_i$ the class $Domain(D)$ is:

$$
\begin{array}{lll}
\{\ add & : & BinaryOp\,(D)\ ;\ assoc\_add : Assoc\,(D, add); \\
\ \ zero & : & D\ ;\ identity\_zero : Identity\,(D, add, zero); \\
\ \ minus & : & D \to D\ ;\ inverse\_minus : Inverse\,(D, add, zero, minus); \\
\ \ mult & : & BinaryOp\,(D)\ ;\ assoc\_mult : Assoc\,(D, mult); \\
\ \ one & : & D\ ;\ identity\_one : Identity\,(D, mult, one); \\
\ \ div & : & BinaryOp\,(D); \\
\ \ rm & : & BinaryOp\,(D)\,;\ \forall x, y : D.(x = y * div(x, y) + rm(x, y)\ in\ D); \\
\ \ discrete & : & \forall x, y : D.(x = y \text{ in } D \text{ or } x \neq y\ in\ D); \\
\ \ less\_eq & : & BinaryRel\,(D); porder : PartialOrder\,(D, less\_eq); \\
\ \ trichot & : & \forall x, y : D.(x \leq y \text{ or } y \leq x) \\
 & & \qquad \text{and } (x \leq y\ \ \&\ \ y \leq x \text{ implies } x = y \text{ in } D); \\
\ \ cong & : & \forall x, y, z : D.(x \leq y \text{ implies } x + z \leq y + z) \text{ and} \\
 & & \qquad (x \leq y \text{ and } z \geq 0 \text{ implies } x * z \leq y * z) \text{ and} \\
 & & \qquad (x \leq y \text{ and } z < 0 \text{ implies } y * z \leq x * z).\ \}
\end{array}
$$

The domain can be made *inductive* if we add an induction principle such as

$$ind : \forall P : D \to Prop_i.\ (P(0)\ \&\ \forall z : \{x : D | x \geq 0\}.P(z) \quad \text{implies} \quad P(z+1)$$

$$\&\ \forall z : \{x : D | x < 0\}.\ P(z+1) \text{ implies } P(z)) \text{ implies } \forall x{:}D.\ P(x).$$

In type theory it is also easy to allow a kind of *primitive recursive definition* over D by generalizing the induction to

$$ind : \forall A : D \to \mathbb{U}_i.A(0) \to (z : \{y : D | y \geq 0\} \to A(z) \to A(z+1)) \to$$

$$(z : \{y : D | y < 0\} \to A(z+1) \to A(z))\forall x : D.A(x).$$

$$ind\_eq : \forall b : A(0).\forall f : (z : \{y : D | y \geq 0\} \to A(z) \to A(z+1)).$$

$$\forall g : z : (\{y : D | y < 0\} \to A(z+1) \to A(z)).$$

$$ind(b)(f)(g)(0) = b\ in\ A(0) \text{ and } \forall y : D(y \geq 0 \text{ implies}$$

$$ind(b)(f)(g)(y) = f(y)(ind(b)(f)(g)(y-1))\ in\ A(y))$$

$$\text{and } \forall y : D.(y < 0 \text{ implies}$$

$$ind(b)(f)(g)(y) = g(y)(ind(b)(f)(g)(y+1))\ in\ A(y)).$$

If we take A to be the type $(D \to D)$, then the induction constructor allows us to define functions $D \to (D \to D)$ by induction. For example, here is a definition of factorial over D. We take $A(x) = D$ for all x, so we are defining

a function from D to D. On input 0, we build 1; in the case for $z > 0$ and element u in $A(z)$, we build the element $z * u$ as result. For $z < 0$, we take 0 as the result. The form of definition is

$$ind(\lambda(x.D))(1)(\lambda(z.\lambda(u.z * u)))(\lambda(z.\lambda(u.0))).$$

11 The axiom of choice

Halmos notes that it has been important to examine each consequence of the axiom of choice to see "the extent to which the axiom is needed in the proof." He said, "an alternative proof without the axiom of choice spelled victory." It was thought that results without the axiom of choice were safer. But in fact, in the computational mathematics used here, which is very safe, one form of the axiom of choice is provable! We start this section with a proof of its simplest form.

If we know that $\forall x\!:\!A.\ \exists y\!:\!B.\ R(x,y)$, then the axiom of choice tells us that there is a function f from A to B such that $R(x, f(x))$ for all x in A. We can state this symbolically as follows (using $\Rightarrow$ for implication):

Axiom of Choice $\forall x\!:\!A.\ \exists y\!:\!B.\ R(x,y) \Rightarrow \exists f\!:\!A \to B.\ \forall x\!:\!A.\ R(x, f(x))$.

Here is a proof of the axiom. We assume $\forall x\!:\!A.\ \exists y\!:\!B.\ R(x,y)$. According to the computational meaning of $\forall x : A$, we know that there is a function g from A to evidence for $\exists y\!:\!B.\ R(x,y)$. The evidence for $\exists y\!:\!B.\ R(x,y)$ is a pair of a witness, say $b(x)$, and evidence for $R(x, b(x))$; call that evidence $r(x)$. Thus the evidence is the pair $< b(x), r(x) >$.

So now we know that on input x from A, g produces $< b(x), r(x) >$. We can define f to be $\lambda(x.\ b(x))$. We know that $f\ in\ A \to B$ since $b(x)\ in\ B$ for any $x\ in\ A$. So the witness to $\exists f : A \to B$ is now known. Can we also prove $\forall x : A.\ R(x, f(x))$? For this we need a function, say h, which on input x produces a proof of $R(x, f(x))$. We know that $\lambda(x.r(x))$ is precisely this function. So the pair we need for the conclusion is

$$< \lambda(x.b(x)), \lambda(x.r(x)) >$$

where $b(x) = 1of(g(x))$ and $r(x) = 2of(g(x))$. Thus the implication is exhibited as

$$\lambda(g. < \lambda(x.1of(g(x))), \lambda(x.2of(g(x))) >).$$

This is the computational content of the axiom of choice.

In set theory the corresponding statement of the axiom of choice is the statement that the product of a family of sets $B(x)$ indexed by A is nonempty if and only if each $B(x)$ is. That is:

$$\Pi x : A.\ B(x) \text{ is inhabited}$$
$$\text{iff}$$
$$\text{for each } x \text{ in } A,\ B(x) \text{ is inhabited.}$$

We can state this as a special case of our axiom by taking $R(x, y)$ to be trivial, say *True.*

$$\exists f : (x : A \to B(x)).\, True \text{ iff } \forall x{:}A.\, \exists y{:}B(x).\, True.$$

Another formulation of the axiom in set theory is that for any collection $\mathcal{C}$ of nonempty subsets of a set A, there is a function f taking $x \in \mathcal{C}$ to an element $f(x)$ in x. Halmos states this as

$$\exists f.\, \forall x : (\mathcal{P}(A) - \{\emptyset\}).\, f(x) \in x$$

We can almost say this in type theory as: there is an element of the type

$$x : \{Y : \mathcal{P}(A) \,|\, Y \text{ is nonempty}\} \to x.$$

One problem with this formulation is that $\mathcal{P}(A)$, the type of all subsets of A, does not exist. The best we can do, as we discussed in the section on power sets, is define

$$\mathcal{P}_i(A) = \{x : \mathbb{U}_i \,|\, x \sqsubseteq A\}.$$

If we state the result as the claim that

$$x : (Y : \mathcal{P}_i(A) \times Y) \to x$$

is inhabited, then it is trivially true since the choice function f takes as input a type Y, that is, a subtype of A and an element $y \in Y$, and it produces the y, e.g.

$$f(< Y, y >) = y$$

so $f(x) = 2of(x)$.

But this simply shows that we can make the problem trivial. In a sense our statements of the axiom of choice have made it too easy.

Another formulation of the axiom of choice would be this:

$$\text{Set Choice} \quad x : \{y : \mathcal{P}_i(A) | y\} \to x.$$

Recall that the evidence for $\{x{:}A|B\}$ is simply an object $a\, in\, A$; the evidence for B is suppressed. That is, we needed it to show that a is in $\{x{:}A|B\}$, but then by using the subtype, we agree not to reveal this evidence. Set Choice says that if we have an arbitrary subtype x of A which is inhabited but we do not know the inhabitant, then we can recover the inhabitant uniformly.

This Set Choice is quite unlikely to be true in type theory. We can indeed make a recursive model of type theory in which it is false. This is perhaps the closest we can get in type theory to stating the axiom of choice in its classical sense, and this axiom is totally implausible. Consider this version:

$$P : \{p : Prop_i | p \vee \neg p\} \to P \vee \neg P.$$

We might call this "propositional choice." It is surely quite implausible in computational logic.

12 Computational complexity

As Turing and Church showed, computability is an abstract mathematical concept that does not depend on physical machines (although its practical value and its large intellectual impact do depend very much on physical machines, and on the sustained steep exponential growth in their power). Computability can be axiomatized as done in these notes by underpinning mathematical objects with data having explicit structure, and by providing reduction rules for destructor operations on data.

Hartmanis and Stearns showed that the cost of computation can also be treated mathematically, even though it might at first seem that this would depend essentially on characteristics of physical machines, such as how much *time* an operation required, or how much circuitry or how much electrical power or how much bandwidth, etc. It turns out that our abstract internal characterizations of resource expenditure during computation is correlated in a meaningful way with actual physical costs. We call the various internal measures *computational complexity measures.*

Although we can define computational complexity mathematically, it is totally unlike all the ideas we have seen in the first eleven sections. Moreover, there is no comparably general account of computational complexity in set theory, since not all operations of set theory have computational meaning.

Let's start our technical story by looking at a simple example. Consider the problem of computing the *integer square root* of a natural number, say $root(0) = 0$, $root(2) = 1$, $root(4) = 2$, $root(35) = 5$, $root(36) = 6$, etc. We can state the problem as a theorem to be proved in our computational logic:

Root Theorem $\forall n\colon \mathbb{N}.\ \exists r\colon \mathbb{N}.\ r^2 \leq n < (r+1)^2$.

We prove this theorem by induction on n. If n is zero, then taking r to be 0 satisfies the theorem.

Suppose now that we have the root for n; this is our induction hypothesis, namely

$$* \exists r\colon \mathbb{N}.\ r^2 \leq n < (r+1)^2.$$

Let r_0 be this root. Our goal is the find the root for $n+1$. As in the case of $root(34) = 5$ and $root(35) = 5$ and $root(36) = 6$, the decision about whether the root of $n+1$ is r_0 or r_0+1 depends precisely on whether $(r_0+1)^2 \leq n+1$. So we consider these two cases:

1. Case $n+1 < (r_0+1)^2$; then since ${r_0}^2 \leq n$, we have:

   $$ {r_0}^2 \leq n+1 < (r_0+1)^2. $$

   Hence, r_0 is the root of $n+1$.

2. Case $(r_0+1)^2 \leq n+1$. Notice that since $n < (r_0+1)^2$, it follows that:

$$n+1 < (r_0+1)^2+1 \text{ and } ((r_0+1)+1)^2 > (r_0+1)^2+1.$$

Hence r_0+1 is the root of $n+1$. This ends the proof.

By the axiom of choice, there is a function $root \in \mathbb{N} \to \mathbb{N}$ such that

$$\forall n{:}\mathbb{N}.\ root(n)^2 \leq n < (root(n)+1)^2.$$

Indeed, we know the code for this function because it is derived from the computational meaning of the proof. It is this recursive function:

$$\begin{array}{rcl} root(0) & = & 0. \\ root(n+1) & = & \mathit{let}\ r_0 = root(n) \\ & & \mathit{in}\ \mathit{if}\ n+1 < (r_0+1)^2\ \mathit{then}\ r_0 \\ & & \mathit{else}\ r_0+1 \\ & & \mathit{end}. \end{array}$$

It's easy to determine the number of computation steps needed to find the root of n. Basically it requires $4 \cdot root(n)$. This is a rather inefficient computation. It is basically the same as the cost of the program:

$$\begin{array}{l} r := 0 \\ \mathit{while}\ r^2 \leq n\ \mathit{do} \\ r := r+1 \\ \mathit{end}. \end{array}$$

There are worse computations that look similar, such as

$$\begin{array}{rl} slow\_root(n) = & \mathit{if}\ n < 0\ \mathit{then}\ 0 \\ & \mathit{else}\ \mathit{if}\ n < (slow\_root(n-1)+1)^2 \\ & \mathit{then}\ slow\_root(n-1) \\ & \mathit{else}\ slow\_root(n-1)+1. \end{array}$$

This computation takes $4 \cdot 2^{root(n)}$.

We might call *slow_root* an "exponential algorithm," but usually we measure computational complexity in terms of the length of the input, which is essentially $\log(n)$. So even $4 \cdot root(n)$ is an exponential algorithm. It is possible to compute $root(n)$ in time proportional to $\log(n)$ if we take large steps as we search for the root. Instead of computing $root(n-1)$, we look at $root(n \div 4)$. This algorithm will call *root* at most $\log(n)$ times. Here is the algorithm:

$$\begin{array}{rl} sqrt(x) = & \mathit{if}\ x = 0\ \mathit{then}\ 0 \\ & \mathit{else}\ \mathit{let}\ r = sqrt(x \div 4) \\ & \mathit{in}\ \mathit{if}\ x < (2*r+1)^2\ \mathit{then}\ 2*r \\ & \mathit{else}\ 2*r+1 \\ & \mathit{end}. \end{array}$$

This algorithm comes from the following proof. It uses the following *efficient-induction* principle:

If $P(0)$ and if for $y\, in\, \mathbb{N},\ P(y \div 4)$ implies $P(y)$, then $\forall x{:}\mathbb{N}.\ P(x)$.

Fast Root Theorem $\forall x{:}\mathbb{N}.\ \exists r{:}\ \mathbb{N}.\ r^2 \leq x < (r+1)^2$.

Proceed by efficient induction. When $x = 0$, take $r = 0$. Otherwise, assume $\exists r : \mathbb{N}.r^2 \leq x \div 4 < (r+1)^2$. Now let r_0 be the root assumed to exist and compare $(2 * r_0 + 1)^2$ with x.

1. Case $x < (w * r_0 + 1)^2$; then since ${r_0}^2 \leq x \div 4$, we know that

$$(2 * r_0)^2 \leq x.$$

   So $2 \cdot r$ is the root of x.

2. Case $(2 * r_0 + 1)^2$. Then we know that:

$$(2 * r_0 + 2)^2 = (2 \cdot (r_1))^2 = 4 \cdot (r+1)^2 > x,$$

   since

$$x \div 4 < (r_0 + 1)^2 \text{ and } 4 \cdot (x \div 4) < 4 \cdot (r_0 + 1)^2.$$

So $2 \cdot r_0 + 1$ is the root of x. This ends the proof.

If we use binary (or decimal) notation for natural numbers and implement the basic operations of addition, subtraction, multiplication and integer division efficiently, then we know that this algorithm operates in number of steps $0(\log(x))$. This is a reasonably efficient algorithm.

The question we want to explore is how to express this basic fact about runtime of *sqrt* inside the logic. Our observation that the runtime is $0(\log(x))$ is made in the *metalogic*, where we have access to the computation rules and the syntax of the algorithm. Inside the logic we do not have access to these aspects, and we cannot easily extend our rules to include them because these rules conflict with other more basic decisions. Let's look at this situation more carefully.

In the logic, the three algorithms *root*, *slow_root*, and *sqrt* are functions from $\mathbb{N}$ to $\mathbb{N}$, and as functions they are equal, because $slow\_root(x) = sqrt(x)$ for all $x\, in\, \mathbb{N}$. Thus $slow\_root = sqrt\, in\, \mathbb{N} \to \mathbb{N}$. This fact about equality means that we cannot have a function $Time$ such as

$$Time(slow\_root)(x) = 4 * 2\, root(x)$$

and

$$Time(sqrt)(x) = 0(\log(x)),$$

because a function such as $Time$ must respect equality; so if $f = g$ then $Time(f) = Time(g)$.

At the metalevel we are able to look at *slow_root* and *sqrt* as terms rather than as functions. This is the nature of the metalogic; it has access to syntax and rules. So our exploration leads us to ask whether we can somehow express facts about the metalogic inside the object logic. Gödel showed one way to do this, by "encoding" terms and rules as numbers by the mechanism of *Gödel numbering*.

We propose to use a mechanism more natural than Gödel numbering; we will add the appropriate metalogical types and rules into the logic itself. We are interested in these components of the metalogic:

Metalogic	Object Logic
term	Term
x *evalsto* y in m steps	x $EvalsTo$ y in m Steps
$eval(x)$	$Eval(x)$

The idea is that we create a new type, called $Term$. We show in the metalogic that $Term$ represents $term_2$ in that for each $t \in term$, there is an element $rep(t)\, in\, Term$. If $t = t'\, in\, term$, then $rep(t) = rep(t')\, in\, Term$.

If t evaluates to $t'\, in\, term$, then $rep(t)\ EvalsTo\ rep(t')\ in\ Term$, and if $eval(t) = t'$, then $Eval(rep(t)) = rep(t')$.

We will also introduce a function, $ref(t)$, which provides the meaning of elements of $Term$. So for a closed $term\ t\ in\ Term$, $ref(t)$ will provide its meaning as an element of a type A of the theory, if it has such a meaning. For each type A, there will be the collection of $Terms$ that represent elements of A, denoted $[A] = \{x : Term | \exists y : A. ref(x) = y\ in\ A\}$. The relation $ref(x)\ in\ A$ is a proposition $Term \times A \rightarrow Prop$.

For each type A we can define a collection of $Terms$ that denote elements of A. Let

$$[A] = \{x : Term | \exists y : A. ref(x) = y\ in\ A\}.$$

The relation $ref(x)\ is\_in\ A$ is defined as

$$\exists y : A.\ ref(x) = y\ in\ A.$$

This is a propositional function on $Term \times A$.

Now given an element of $[A]$, we can measure the number of steps that it takes to reduce it to canonical form. This is done precisely as in the meta theory, using the relation $e\ EvalsTo\ e'$ in n steps. Having $Term$ as a type makes it possible to carry over into the type theory the evaluation relation and the step counting measure that is part of it. We can also define other resource measures as well, such as the amount of *space* used during a computation. For this purpose we could use the size of a term as the basic building block for a space measure.

Once we have complexity measures defined on *Term*, we can define the concept of a complexity class, as follows.

The evaluation relation on *Term* provides the basis for defining computational complexity measures such as time and space. These measures allow us to express traditional results about complexity classes as well as recent results concerning complexity in higher types. The basic measure of time is the *number of evaluation steps to canonical form.* Here is a definition of the notion that e runs within time t:

$$\begin{aligned} \mathit{Time}(e,t)\ \mathit{iff}\ \exists n{:}[0\cdots t].\,\exists f{:}[0\cdots n]\to \mathit{Term}.\ & f(0)=e\ \mathit{in}\ \mathit{Term}\ \wedge \\ & \mathit{iscanon}(f(n))=\mathit{true}\ \mathit{in}\ \mathbb{B}\ \wedge \\ & \forall i{:}[0\cdots n-1]. \\ & \qquad f(i)\ \mathit{EvalsTo}\ f(i+1). \end{aligned}$$

We may define a notion of space in a similar manner. First, we may easilynoindent **d**efine a function *size* with type $\mathit{Term}\to\mathbb{N}$ which computes the number of operators in a term. Then we define the predicate $\mathit{Space}(e,s)$, that states that e runs in space at most s:

$$\begin{aligned} \mathit{Space}(e,s)\ \mathit{iff}\ \exists n{:}\mathbb{N}.\,\exists f{:}[0\cdots n]\to & \\ \mathit{Term}.\ & f(0)=e\ \mathit{in}\ \mathit{Term}\ \text{and} \\ & \mathit{iscanon}(f(n))=\mathit{true}\ \mathit{in}\ \mathbb{B}\ \mathit{and} \\ & \forall i{:}[0\cdots n-1].\ f(i)\ \mathit{EvalsTo}\ f(i+1)\ \mathit{and} \\ & \forall i{:}[0\cdots n].\ \mathit{size}(f(i))\le s. \end{aligned}$$

Using these, we may define the resource-indexed type $[T]^t_s$ of terms that evaluate (to a member of T) within time t and space s:

$$[T]^t_s \stackrel{\text{def}}{=} \{e:[T] \mid \mathit{Time}(e,t)\ \mathit{and}\ \mathit{Space}(e,s)\}$$

One interesting application of the resource-indexed types is to define types like Parikh's *feasible numbers,* numbers that may be computed in a "reasonable" time. Benzinger shows another application.

With time complexity measures defined above, we may define complexity classes of functions. Complexity classes are expressed as function types whose members are required to fit within complexity constraints. We call such types *complexity-constrained* function types. For example, the quadratic time, polynomial time, and polynomial space computable functions may be defined as follows:

$$\begin{aligned} & \mathit{Quad}(x{:}A\longrightarrow B)\ = \\ & \qquad \{f:[x{:}A\to B] \mid \exists c{:}\mathbb{N}.\,\forall a{:}[A]^0.\ \mathit{Time}(\langle\!\langle \mathit{app}^*, f, a\rangle\!\rangle, c\cdot \mathit{size}(a)^2)\} \\ & \mathit{Poly}(x{:}A\longrightarrow B)\ = \\ & \qquad \{f:[x{:}A\to B] \mid \exists c,c'{:}\mathbb{N}.\,\forall a{:}[A]^0.\ \mathit{Time}(\langle\!\langle \mathit{app}^*, f, a\rangle\!\rangle, c\cdot \mathit{size}(a)^{c'})\} \\ & \mathit{PSpace}(x{:}A\longrightarrow B)\ = \end{aligned}$$

$$\{f : [x{:}A \to B] \mid \exists c, c'{:}\mathbb{N}.\, \forall a{:}[A]^0.\, \mathit{Space}(\langle\!\langle \mathit{app}^*, f, a\rangle\!\rangle, c \cdot \mathit{size}(a)^{c'})\}$$

One of the advantages of constructive logic is that when the existence of an object is proven, that object may be constructed, as we saw in our discussion of the axiom of choice, where a computable function is constructed from a proof. However, there is no guarantee that such functions may feasibly be executed. This has been a serious problem in practice, as well as in principle.

Using the complexity-constrained functions, we may define a *resource-bounded logic* that solves this problem. As we noted in Section 9 under the propositions-as-types principle, the universal statement $\forall x : A.\ B$ corresponds to the function space $x{:}A{\to}B$. By using the complexity-constrained function space instead, we obtain a *resource-bounded* universal quantifier. For example, let us denote the quantifier corresponding to the polynomial-time computable functions by $\forall_{poly} x{:}A.\ B$. By proving the statement $\forall_{poly} x{:}A.\, \exists y{:}B.\, P(x, y)$, we guarantee that the appropriate y may actually be feasibly computed from a given x.

The following is a proposition expressing the requirement for a *feasible integer square root:*

$$\forall_{poly} x : \mathbb{N}.\ \exists r : \mathbb{N}.\ \{r^2 \leq x < (r + 1^2\}.$$

Bibliographic notes

Section 1 — Types and Equality

The Halmos book [57] does not cite the literature since his account is of the most basic concepts. I will not give extensive references either, but I will cite sources that provide addtional references.

One of the best books about basic set theory, in my opinion, is still *Foundations of Set Theory,* by Fraenkel, Bar-Hillel and Levy [50].

There are a few text books on basic type theory. The 1986 book by the PRL group [34] is still relevant, and it is now freely available at the Nuprl Web site (www.cs.cornell.edu/Info/Projects/NuPrl/). Two other basic texts are *Type Theory and Functional Programming,* by Thompson [103], and *Programming in Martin-Löf's Type Theory* [86], by Nordstrom, Petersson and Smith. A recent book by Ranta, *Type-theoretical Grammar* [93], has a good general account of type theory. Martin-Löf type theory is presented in his *Intuitionistic Type Theory* [80] and *Constructive Mathematics and Computer Programming* [79].

Section 2 — Subtypes and Set Types

The notion of subtype is not very thoroughly presented in the literature. There is the article by Constable and Hickey [39] which cites the basic literature. Another key paper is by Pierce and Turner [91]. The PhD theses from Cornell and Edinburgh deal with this subject, especially Crary [46], Hickey [63], and Hofmann [65].

Section 3 — Pairs

Cartesian products are standard, even in the earliest type theories, such as Curry [47] and deBruijn [49].

Section 4 — Union and Intersection

The intersection type is deeply studied in the lambda calculus. See the papers of Coppo and Dezani-Ciancaglini [42], Compagnoni [33], and Pierce [92].

The logical role of intersection is discussed by Caldwell [26, 25] and earlier by Allen [4]. The newest results are from Kopylov [71] and Girard [54].

Section 5 — Functions and Relations

This section is the heart of the untyped and typed lambda calculus. See Barendregt [7, 8], Stenlund [102], and Church [29]. The treatment of relations goes back to Frege and Russell and is covered well in Church [30].

Section 6 — Universes, Powers and Openness

The account of universes is from Per Martin-Löf [79] and is informed by Allen [4] and Palmgren [88]. Insights about power sets can be found in Fraenkel et al. [50], Beeson [11], and Troelstra [105].

Section 7 — Families

Families are important in set theory, and accounts such as Bourbaki [18] inform Martin-Löf's approach [79].

Section 8 — Lists and Numbers

List theory is the basis of McCarthy's theory of computing [82]. The Boyer-Moore prover was used to create an extensive formal theory [19], and the Nuprl libraries provide a constructive theory.

Section 9 — Logic and the Peano Axioms

Our approach to logic comes from Brouwer as formalized in Heyting [61]. One of the most influential accounts historically is Howard [66] and also deBruijn [48, 49] for Automath. The Automath papers are collected in [85].

The connection between propositions and types has found its analogue in set theory as well. The set theory of Anthony P. Morse from 1986 equated sets and propositions. He "asserted a set" by the claim that it was non-empty. Morse believed "that every (mathematical) thing is a set." For him, conjunction is intersection, disjunction is union, negation is complementation. Quantification is the extension of these operations to families.

"Each set is either true or false, and each sentence is a name for a set."

Section 10 — Structures, Records and Classes

The approach to records and classes developed here is based entirely on the work of Constable, Hickey and Crary. The basic papers are [39, 46]. The account by Betarte and Tasistro [15] is related. There is an etensive literature cited in the books of Gunter and Mitchell [56]. The treatment of inductive classes is based on Basin and Constable [9].

Section 11 — The Axiom of Choice

There are many books about the axiom of choice. One of the best is Fraenkel et al.[50] Another is Gregory Moore's *Zermelo's Axiom of Choice : Its Origins, Development, and Influence* [83]. Our account is based on Martin-Löf [79].

Section 12 — Computational Complexity

The fundamental concepts and methods of computational complexity theory were laid down in the seminal paper of Hartmanis and Stearns, *On the Computational Complexity of Algorithms* [60]. Many textbooks cover this material, for example [75]. The extension of this theory to higher-order objects is an active field [99], and the study of feasible computation is another active area related to this article [12, 69, 72, 73, 74]. These topics are covered also in Schwichtenberg [13], and in the articles of Jones [70], Schwichtenberg [98], and Wainer [87] in this book.

The work reported here is new and based largely on Constable and Crary [38] and Benzinger [14], as well as examples from Kreitz and Pientka [90].

One interesting application of the resource-indexed types is to define types like Parikh's feasible numbers [89], numbers that may be computed in a "reasonable" time. Benzinger [14] shows another application.

Acknowledgements

I would like to thank Juanita Heyerman for preparing the manuscript so carefully and for contributing numerous suggestions to improve its form. I also want to thank the excellent students at the Marktoberdorf summer school for encouraging me to write from this point of view.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[2] Martin Abadi and Leslie Lamport. Conjoining specifications. *ACM Toplas*, 17(3), May 1995.

[3] Peter Aczel. Notes on the simply typed lambda calculus. In Ulrich Berger et al., editor, *Computational Logic. Proceedings of the NATO ASI, Marktoberdorf, Germany, July 29–August 10, 1997*, volume 165 of *NATO ASI Ser., Ser. F, Comput. Syst. Sci.*, pages 57–97, Berlin, 1999. Springer.

[4] Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.

[5] R. C. Backhouse. Constructive type theory–an introduction. In M. Broy, editor, *Constructive Methods in Computer Science,* NATO ASI Series, Vol. F55*: Computer & System Sciences*, pages 6–92. Springer-Verlag, New York, 1989.

[6] R. C. Backhouse, P. Chisholm, G. Malcolm, and E. Saaman. Do-it-yourself type theory (part I). *Formal Aspects of Computing*, 1:19–84, 1989.

[7] Henk P. Barendregt. The typed lambda calculus. In Jon Barwise, editor, *Handbook of Mathematical Logic*, pages 1091–1132. North-Holland, NY, 1977.

[8] Henk P. Barendregt. *Handbook of Logic in Computer Science*, volume 2, chapter Lambda Calculi with Types, pages 118–310. Oxford University Press, 1992.

[9] David A. Basin and Robert L. Constable. Metalogical frameworks. In G. Huet and G. Plotkin, editors, *Logical Environments*, chapter 1, pages 1–29. Cambridge University Press, Great Britain, 1993.

[10] J. L. Bates and R. L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):53–71, 1985.

[11] Michael J. Beeson. *Foundations of Constructive Mathematics*. Springer-Verlag, 1985.

[12] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.

[13] Stephen J. Bellantoni, Karl-Heinz Niggl, and Helmut Schwichtenberg. Higher type recursion, ramification and polynomial time. *Annals of Pure Applied Logic*, 104(1-3):17–30, 2000.

[14] Ralph Benzinger. Automated complexity analysis of Nuprl extracted programs. *Journal of Functional Programming*, 2000. To Appear.

[15] Gustavo Betarte and Alvaro Tasistro. Extension of Martin Löf's type theory with record types and subtyping. chapter 2, pages 21–39, In Twenty-Five Years of Constructive Type Theory. Oxford Science Publications, 1999.

[16] Mark Bickford and Jason Hickey. An object-oriented approach to verifying group communication systems. Department of Computer Science, Cornell University, Unpublished, 1998.

[17] N. Bourbaki. *Elements of Mathematics, Algebra, Volume 1.* Addison-Wesley, Reading, MA, 1968.

[18] N. Bourbaki. *Elements of Mathematics, Theory of Sets.* Addison-Wesley, Reading, MA, 1968.

[19] R. S. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

[20] R. S. Boyer and J. S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science*, pages 103–84. Academic Press, New York, 1981.

[21] L. E. J. Brouwer. *Collected Works* A. Heyting, ed., volume 1. North-Holland, Amsterdam, 1975.

[22] K. B. Bruce and G. Longo. A modest model of records, inheritance, and bounded quantification. In J. C. Mitchell C. A. Gunter, editor, *Theoretical Aspects of Object-Oriented Programming, Types, Semantics and Language Design*, chapter III, pages 151–196. MIT Press, Cambridge, MA, 1994.

[23] Kim Bruce and John Mitchell. PER models of subtyping, recursive types and higher–order polymorphism. *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Lanuages*, pages 316–327, January 1992.

[24] S. Buss. The polynomial hierarchy and intuitionistic bounded arithmetic. In *Structure in Complexity Theory,* Lecture Notes in Computer Science. 223, pages 77–103. Springer, Berlin, 1986.

[25] J. Caldwell, I. Gent, and J. Underwood. Search algorithms in type theory. *Theoretical Computer Science*, 232(1–2):55–90, February 2000.

[26] James Caldwell. Moving proofs-as-programs into practice. In *Proceedings of the 12th IEEE International Conference on Automated Software Engineering.* IEEE Computer Society, 1997.

[27] Luca Cardelli. Extensible records in a pure calculus of subtyping. In Carl. A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics and Language Design.* MIT Press, 1994. A preliminary version appeared as SRC Research Report No. 81, 1992.

[28] Luca Cardelli and John Mitchell. Operations on records. In J. C. Mitchell C. A. Gunter, editor, *Theoretical Aspects of Object-Oriented Programming, Types, Semantics and Language Design*, chapter IV, pages 295–350. MIT Press, Cambridge, MA, 1994.

[29] Alonzo Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematical Studies*. Princeton University Press, Princeton, 1951.

[30] Alonzo Church. *Introduction to Mathematical Logic, Vol. I.* Princeton University Press, 1956.

[31] A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Proceedings of the 1964 International Conference for Logic, Methodology, and Philosophy of Science*, pages 24–30. North-Holland, 1965.

[32] Adriana B. Compagnoni and Benjamin C. Pierce. Higher-order intersection types and multiple inheritance. *Math. Struct. in Comp. Science*, 11:1–000, 1995.

[33] Adriana Beatriz Compagnoni. *Higher-Order Subtyping with Intersection Types*. PhD thesis, Katholieke Universiteit Nijmegen, January 1995.

[34] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.

[35] Robert L. Constable. Using reflection to explain and enhance type theory. In Helmut Schwichtenberg, editor, *Proof and Computation*, volume 139 of *NATO Advanced Study Institute, International Summer School held in Marktoberdorf, Germany, July 20-August 1, NATO Series F*, pages 65–100. Springer, Berlin, 1994.

[36] Robert L. Constable. Experience using type theory as a foundation for computer science. In *Proceedings of the Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 266–279. LICS, June 1995.

[37] Robert L. Constable. Types in logic, mathematics and programming. In S. R. Buss, editor, *Handbook of Proof Theory*, chapter X, pages 684–786. Elsevier Science B.V., 1998.

[38] Robert L. Constable and Karl Crary. Computational complexity and induction for partial computable functions in type theory. In W. Sieg, R. Sommer, and C. Talcott, editors, *Reflections on the Foundations of Mathematics: Essays in Honor of Solomon Feferman*, Lecture Notes in Logic, pages 166–183. Association for Symbolic Logic, 2001. 2001.

[39] Robert L. Constable and Jason Hickey. Nuprl's Class Theory and its Applications. In Friedrich L. Bauer and Ralf Steinbrueggen, editors, *Foundations of Secure Computation*, NATO ASI Series, Series F: Computer & System Sciences, pages 91–116. IOS Press, 2000.

[40] Robert L. Constable and Scott F. Smith. Computational foundations of basic recursive function theory. *Theoretical Computer Science*, 121:89–112, December 1993.

[41] Robert L. and Paul B. Jackson Constable, Pavel Naumov, and Juan Uribe. Constructively formalizing automata. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 213–238. MIT Press, Cambridge, 2000.

[42] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ–calculus. *Notre-Dame Journal of Formal Logic*, 21(4):685–693, October 1980.

[43] Thierry Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[44] Karl Crary. Foundations for the implementation of higher-order subtyping. In *1997 ACM SIGPLAN International Conference on Functional Programming*, Amsterdam, June 1997. to appear.

[45] Karl Crary. Simple, efficient object encoding using intersection types. Technical Report TR98-1675, Department of Computer Science, Cornell University, April 1998.

[46] Karl Crary. *Type–Theoretic Methodology for Practical Programming Languages*. PhD thesis, Cornell University, Ithaca, NY, August 1998.

[47] H. B. Curry, R. Feys, and W. Craig. *Combinatory Logic*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1958.

[48] N. G. deBruijn. Set theory with type restrictions. In A. Jahnal, R. Rado, and V. T. Sos, editors, *Infinite and Finite Sets*, volume I of *Collections of the Mathematics Society*, pages 205–314. J. Bolyai 10, 1975.

[49] N. G. deBruijn. A survey of the project Automath. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

[50] A. A. Fraenkel, Y. Bar-Hillel, and A. Levy. *Foundations of Set Theory*, volume 67 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 2nd edition, 1984.

[51] J-Y. Girard. On the meaning of logical rules II: Multiplicatives and additives. In F. L. Bauer and R. Steinbrueggen, editors, *Foundations of Secure Computing*, pages 183–212. IOS Press, Berlin, 2000.

[52] J-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge Tracts in Computer Science, Vol. 7. Cambridge University Press, 1989.

[53] Jean-Yves Girard. On the meaning of logical rules. i: Syntax versus semantics. In Ulrich Berger et al., editor, *Computational Logic. Proceedings of the NATO ASI, Marktoberdorf, Germany, July 29–August 10, 1997*, volume 165 of *NATO ASI Ser., Ser. F, Comput. Syst. Sci.*, pages 215–272, Berlin, 1999. Springer.

[54] Jean-Yves Girard. Ludics. In *this volume*. Kluwer, 2002.

[55] F. Giunchiglia and A. Smaill. Reflection in constructive and non-constructive automated reasoning. In H. Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 123–140. MIT Press, Cambridge, Mass., 1989.

[56] C. A. Gunter and J. C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming, Types, Semantics and Language Design*. Types, Semantics, and Language Design. MIT Press, Cambridge, MA, 1994.

[57] Paul R. Halmos. *Naive Set Theory*. Springer-Verlag, New York, 1974.

[58] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. *Twenty-First ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 123–137, January 1994.

[59] Juris Hartmanis. *Feasible Computations and Provable Complexity Properties*. SIAM, Philadelphia, PA, 1978.

[60] Juris Hartmanis and R. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematics Society*, 117:285–306, 1965.

[61] A. Heyting. *Intuitionism*. North-Holland, Amsterdam, 1971.

[62] Jason Hickey. Formal objects in type theory using very dependent types. *Foundations of Object-Oriented Languages*, 3, 1996.

[63] Jason Hickey. *The Metaprl Logical Programming Environment*. PhD thesis, Cornell University, Ithaca, NY, 2000.

[64] C. A. R. Hoare. Notes on data structuring. In *Structured Programming*. Academic Press, New York, 1972.

[65] Martin Hofmann. *Extensional Concepts in Intensional Type Theory*. PhD thesis, University of Edinburgh, 1995.

[66] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, NY, 1980.

[67] Douglas J. Howe. Importing mathematics from HOL into Nuprl. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125, of Lecture Notes in Computer Science, pages 267–282. Springer-Verlag, Berlin, 1996.

[68] Douglas J. Howe. Semantic foundations for embedding HOL in Nuprl. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of Lecture Notes in Computer Science, pages 85–101. Springer-Verlag, Berlin, 1996.

[69] N. Immerman. Languages which capture complexity classes. *SIAM Journal of Computing*, 16:760–778, 1987.

[70] Neil D. Jones. Computability and complexity from a programming perspective. In *this volume*. Kluwer, 2002.

[71] Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. Technical Report TR2000–1809, Cornell University, Ithaca, New York, August 2000.

[72] D. Leivant. Finitely stratified polymorphism. *Information and Computation*, 93(1):93–113, July 1991.

[73] Daniel Leivant. Predicative recurrence in finite type. In A. Nerode and Yu V. Matiyasevich, editors, *Logical Foundations of Computer Science, Lecture Notes in Computer Science 813*, pages 227–239. Springer-Verlag, Berlin, 1994.

[74] Daniel Leivant. Ramified recurrence and computational complexity I: Word recurrence and polynomial time. In P. Clote and J. Remmel, editors, *Feasible Mathematics II, Perspectives in Computer Science*. Birkhäuser, 1995.

[75] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, Englewood Cliffs, New Jersey, 1994.

[76] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hi ckey, Mark Hayden, Kenneth Birman, and Robert Constable. Building reliable, high-performance communication systems from compone nts. *Operating Systems Review*, 34(5):80–92, December 1999. Presented at the 17th ACM Symposium on Operating Systems Principles (SOSP'99).

[77] Saunders MacLane and G. Birkhoff. *Algebra*. MacMillan, 1967.

[78] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North-Holland, Amsterdam, 1973.

[79] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North-Holland.

[80] Per Martin-Löf. *Intuitionistic Type Theory. Notes by Giovanni Sambin of a Series of Lectures given in Padua, June 1980.* Number 1 in Studies in Proof Theory, Lecture Notes. Bibliopolis, Napoli, 1984.

[81] Per Martin-Löf. An intuitionistic theory of types. In G. Sambin and J. Smith, editors, *Twenty-Five Years of Constructive Type Theory*, volume 36 of *Oxford Logic Guides*, pages 127–172. Clarendon Press, Oxford, 1998.

[82] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[83] Gregory H. Moore. *Zermelo's Axiom of Choice : Its Origins, Development, and Influence.* Oxford Science Publications. Springer-Verlag, New York, 1982.

[84] P. Naumov. *Formalizing Reference Types in NuPRL.* PhD thesis, Cornell University, August 1998.

[85] R. P. Nederpelt, J. H. Geuvers, and R. C. De Vrijer. *Selected Papers in Automath*, volume 133 of *Studies in Logic and The Foundations of Mathematics.* Elsevier, Amsterdam, 1994.

[86] B. Nordstrom, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory.* Oxford Sciences Publication, Oxford, 1990.

[87] Geoffrey E. Ostrin and Stanley S. Wainer. Proof theory and complexity. In *this volume.* Kluwer, 2002.

[88] Erik Palmgren. On universes in type theory. In G. Sambin and J. Smith, editors, *Twenty-Five Years of Constructive Type Theory*, pages 191–204. Clarendon Press, Oxford, 1998.

[89] R. Parikh. Existence and feasibility in arithmetic. *Jour. Assoc. Symbolic Logic*, 36:494–508, 1971.

[90] Brigitte Pientka and Christoph Kreitz. Instantiation of existentially quantified variables in inductive specification proofs. In *Fourth International Conference on Artificial Intelligence and Symbolic Computation (AISC'98)*, pages 247–258, Kaiserslauten, Germany, October 8–11 1998. LNAI 1476, Springer–Verlag.

[91] B.C. Pierce and D.N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2), 1994.

[92] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, February 1991.

[93] Aarne Ranta. *Type-theoretical Grammar*. Oxford Science Publications. Clarendon Press, Oxford, England, 1994.

[94] J. C. Reynolds. Towards a theory of type structure. In *Proceedings Colloque sur, la Programmation,* Lecture Notes in Computer Science, Vol. 19, pages 408–23. Springer-Verlag, New York, 1974.

[95] J. C. Reynolds. Polymorphism is not set-theoretic. In *Lecture Notes in Computer Science*, pages 145–156, Berlin, 1984. Proceedings International Symposium on Sematics of Data Types, Springer. vol. 173.

[96] Bertrand Russell. Mathematical logic as based on a theory of types. *Am. J. Math.*, 30:222–62, 1908.

[97] V. Yu. Sazonov. On feasible numbers. In *Proceedings of the ASL Meeting*, West Berlin, July 1989.

[98] Helmut Schwichtenberg. Feasible computation with higher types. In *this volume*. Kluwer, 2002.

[99] A. Seth. Turing machine characterizations of feasible functionals of all finite types. In P. Clote and J. Remmel, editors, *Proceedings of MSI Workshop on Feasible Mathematics*, Perspectives in computer science. Birkhauser-Boston, 1994.

[100] Douglas R. Smith. Constructing specification morphisms. *Journal of Symbolic Computation, Special Issue on Automatic Programming*, 16(5-6):571–606, 1993.

[101] Douglas R. Smith. Toward a classification approach to design. *Proceedings of the Fiftieth International Conference on Algebraic Methodology and Software Technology, AMAST'96, LNCS*, pages 62–84, 1996. Springer Verlag.

[102] S. Stenlund. *Combinators, λ–Terms, and Proof Theory.* D. Reidel, Dordrechte, 1972.

[103] S. Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[104] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory.* Cambridge University Press, 1996.

[105] A.S. Troelstra. Realizability. In S.R. Buss, editor, *Handbook of Proof Theory*, volume 137 of *Studies in Logic and the Foundations of Mathematics*, pages 407–473. Elsevier, 1998.